第 1 章　汇编语言介绍

在这章中您将学一些编写汇编语言的基础。内容包括处理器（Processor）、计算机语言（Language）、数制（Number System）及软件开发工具（Software Development Tool）。

对大部分的计算机而言，有一个非正式的说法，称计算机的心脏是处理器（Processor），你可以在主机板上找到它。在 IBM 兼容计算机上使用的是 Intel X86 家族的处理器。从发明到最近，处理器均是以数字来命名，如 8088、8086、186、286、386、486 等（数字越大代表功能越强）。目前最新、功能最强的处理器，并不是以数字命名，而是称为 Pentium，接下来就是 P6 了。

1.1　机器语言与汇编语言

1.1.1　机器语言

处理器是由许多复杂的芯片组合而成。然而，就我们用户观点来看，所有的处理器只是一个接一个去执行简单指令的机器。

处理器所能执行的指令集合称为指令集，不同的处理器指令集的大小也有所不同。例如 486 处理器指令集就包含 206 条基本指令。而我们在操作系统下，执行一个简单的 BASIC 程序，必须被分解成许多处理器能执行的步骤。

按照指令使用的规则，处理器能直接执行的指令集合称为机器语言。在机器语言中每条指令都是一组数字。一个机器语言程序可以说是由一些非常长的数字字符串组合而成。现在人们已不再直接用机器语言编写程序，而采用较高级的计算机语言，以节省设计程序的时间及工作量。

广义上，我们将计算机语言分为：

（1）高级语言（如 C、C＋＋、BASIC）。

（2）低级语言（汇编语言）。事实上，真正的计算机语言是机器语言（也就是计算机真正能看懂、能执行的语言）。使用低级语言编写的程序需通过编译程序编译成机器语言。

1.1.2　机器指令

一个机器指令当然是一个二进制码数字（也就是只有 0 或 1）。机器指令直接对 CPU 下命令，去执行某一个操作，所以它只对 CPU 有意义。

机器语言指令集只对某一特定的 CPU 有用。IBM-PC 兼容计算机使用 Intel 系列的 CPU。而在其它指令集，如大机器上的机器语言指令集可能就与 PC 所使用的机器语言指令集不同。

例 1：一个典型的双字节 IBM-PC 机器指令如下所示，第 1 个字节 B0 是操作码（op code），第 2 个字节 05 是操作数。

```
B0        05    ;  MOV   AL，05
↑         ↑
操作码    操作数
```

例 2：B4 05；mov AH，05

例 1 与例 2 的操作数均相同，只有操作码不同。可知不同的机器指令会有不同的操作码。

1.1.3 汇编语言

编译程序最大的用处就是将我们所写的汇编语言程序读入 RAM 中，并产生一个与机器语言对等的程序。用高级语言写程序时，除了要了解语言的指令外，还需要知道如何使用它去产生你需要的结果，而编写汇编语言程序要学的地方则更多。除了上述内容之外，还必须了解计算机的基本构造以及数据如何存储与使用。

表 1-1 为几种处理器的指令集大小：

表 1-1 指令集大小

处 理 器	指令集大小（条）
8088/8086	115
186	126
286	142
386	200
486	206
Pentium	216

汇编语言能帮你解开硬件与软件的奥秘。想要了解计算机硬件与操作系统 Operating System)、AP（Application Programs）如何与操作系统之间工作，就非得从汇编语言下手不可。

每一种计算机及其家族都使用不同的机器指令和汇编语言。本书所介绍的程序均可在 IBM PC、PC/XT、PC/AT、80286、80386、80486 上执行。汇编语言最接近机器语言，也是最接近计算机硬件的语言。在许多要求速度的场合，汇编语言是第一选择。

虽然高级语言能为我们带来许多便利，但同时也有许多限制，而汇编语言却没有这方面的困扰。但汇编语言的自由却需付出代价，需要程序员谨慎地处理许多细节。

有些高级语言所不能做到的工作，可将汇编语言写成一个子程序而直接调用此子程序替你工作。或许您用 C 或 Pascal 进行一些屏幕显示操作觉得不够快时，您可以利用汇编语言替你加快这项工作。

1.1.4 汇编语言指令

```
mov  AX，BX
```

上例中汇编语言指令意义是将寄存器 BX 的内容送到寄存器 AX 内。所以可看出 AX 是目的运算符，BX 是源运算符。执行完 mov AX，BX 指令后，BX 内容并不改变。

我们可以说每一条汇编语言指令都会对应或关联一条机器指令。而高级语言，如 C、Pascal 或 Basic，就须靠编译程序翻译成许多机器指令，效率上就差多了。

1.1.5 机器语言与汇编语言

大家都知道高级语言会将我们写的源程序(source code)通过编译程序翻译成机器懂的语言。而汇编语言也有编译程序（Assembler）将程序编译成机器语言，只不过我们用汇编语言写的指令在编译时效率会更高。

1.2 处理器与协处理器

除了处理器（Processor）外，在主机板上可能还有协处理器（Coprocessor）。例如数学协处理器可协助处理器执行数学运算，包含浮点数运算，而且执行速度相当快。虽然计算机可以用软件模拟，但却要牺牲一点时间。

486 和 Pentium 级的处理器一般都内置有数学浮点运算器（486 SX 没有）。使用数学浮点运算器是一个复杂的主题，并不在本书介绍范围内。表 1-2 为数学协处理器指令集大小。

表 1-2 数学协处理器指令集大小

处 理 器	数学协处理器	指令集大小（条）
8088/8088	8087	77
186	8087	77
286	80287	74
386	80387	80
486	内置	80
Pentium	内置	80

虽然 386、486 和 Pentium 比早期的处理器复杂，但用汇编语言设计程序却和在 8088 上一样。以前在 8088 上写的程序在较新的处理器上也同样能顺利执行。

早期的处理器（如 8088、8086、80186）有一个最大的限制是不能使用超过 1MB 内存。286 使用两种不同的模式来解决这个问题。

（1）实模式（Real mode） 执行时就和在 8088、8086、80186 一样，对所有在原始 8088 结构下的 DOS 程序完全兼容。

（2）保护模式（Protected mode） 有 3 个最重要的改进。

1）可使用 16MB 内存。

2）可使用外部磁盘空间来提供 1GB 的虚拟内存（Virtual memory）去模拟内存。

3）提供硬件多任务并行，能快速从一个程序切换到另一个程序。

386 有一最重要的突破就是增加了一个虚拟 86 模式（Virtual 86 mode），它可允许处理器在同一时间执行多个程序，且每一个程序就像在它自己的 8086 机器上一样。除此之外，386 在保护模式下可使用到 4GB 的真实内存（虽然真实硬件并没有提供至此）和 64TB 的虚拟内存。

如果使用一个特别的控制程序，如 Windows，你也可以使 386 在同一时间工作如同多重 DOS 一样。而目前最著名的多任务系统如 OS/2 和 Windows NT 也是使用虚拟 86 模式去提供内置的多重 DOS。

486 和 Pentium 用 cache controller（控制处理器内快速的内存）和内置的数学协处理器去汇编 386 的以上功能。从程序设计的观点来看，我们使用 486 和 Pentium 时就和 386 一样。8088 以后所增加的保护模式的额外指令，除非你要编写高级的系统软件，如操作系统、设备驱动程序或编译器，否则您将不必使用到这些指令。因此几乎所有在 Intel X86 处理器上编写

的程序就如同在 8088 上一样，特别是在编写 DOS 程序时。

本书的目的是让你对基本汇编语言有一个详细的了解。我们基本上不会使用应用在新的处理器上的指令。如果我们提到时将会指出为什么用新的指令会较好。

1.3 何时要使用汇编语言

因为高级语言比较简单，也具有概念性和可读性，所以在大部分工作中，高级语言是我们第一选择。但如果你碰上一些不能解决的事情时，就非要使用汇编语言不可了。因为它能直接控制处理器，所有在计算机上能做到的事情，汇编它都能做到。尤其在加速程序的执行时，你可以将程序中最浪费时间、最需要时间的地方改用汇编语言编写，再将它们连接在一起即可。

使用汇编语言的另外一个理由是，内存空间是昂贵的，而用汇编语言编写的程序所产生的可执行代码是最小的，这也是它速度快的原因之一。为了节省内存空间，使用汇编语言也是重要的方法之一。

对大部分的人而言，是不需要去学习或使用这样低级的语言，但对一个专业的程序员来说是必需的。当你在阅读计算机杂志时，其中的例子可能都是用汇编语言写成的，因为它比较容易解释处理器如何工作。如果不学习汇编语言的话，你将不能通过阅读去获得最新的信息和技巧，以成为一个最顶尖的程序员。

1.4 操作系统（Operating System）

计算机在执行时，操作系统是主要的控制程序。汇编语言程序员有两种情况下会使用到操作系统。

你必须对操作系统下达你要执行的命令，如 copy 文件，执行一个程序等等。在你的程序中，可能会利用到操作系统所提供的许多功能服务，如 I/O、存取文件或目录。像一些比较低级、复杂的工作，操作系统都会提供许多功能可以调用。

本书是假设你的操作系统是 DOS (Disk Operating System)。如果你是使用其它的操作系统如 OS/2 或 Windows NT，你还是可以使用这本书去学习汇编语言程序设计。如果你的程序有直接调用操作系统的功能时，其中的接口 (interface) 就需要你去查你的操作系统参考手册。

1.5 你需要的软件

要编写一个汇编语言程序，你需要有编辑器 (Editor)、编译程序 (assembler)、链接程序 (Linker)，以及调试器 (Debugger)，一般有两种选择。

1. 你可以使用一个全屏幕程序开发环境如 Programmer's WorkBench (PWB，由 Microsoft 提供) 或 Programmer's Platform (由 Borland 提供)。这种环境综合了你要建立程序或调试程序所需用到的工具。而你所要做的工作在下拉式窗口（或称菜单）中都可找到。

2. 使用文本编辑器（需以 ASCII 码存储），在 DOS 命令行下直接编译、链接、调试。在 DOS 中有 EDIT (5.0 版以后才有)、EDLIN 可进行程序的编辑，不必额外再花钱购买其它的

文本编辑器。

DOS 本身提供了一个叫 DEBUG 的调试工具。使用 DEBUG 的好处在于使用简单、自由以及占用较少的内存，但毫无界面可言。建议使用 MASM 所提供的 CodeView 去调试。

1.6 计算机的数制

1.6.1 位（Bit）和字节（Byte）

从计算机输出的数据是我们很容易看懂的文字、数字或符号。然而对计算机而言，它只不过是一个脉冲；如开与关的信号。而且这些信号所代表的意义都只有两种状态，我们常称为 0 与 1。每个单一的 0 或 1 称为位（bit），位也是计算机表示数据的最小单位。若将 n 个位串起来，就有 n 位，可形成 2^n 种不同的数字。在计算机内任何数据，如数字、字母、符号，都是利用这种位串的方式来表示。而 8 位组成一个字节（byte）如表 1-3 所示。而字（word）则须视不同的处理器而有不同的大小，在 80286 中，1 word＝16 bit。

表 1-3

位　数	名　称
8	byte
16	word（2 bytes）
32	double word（4 bytes）
64	quadword（8 bytes）
128	paragraph（16 bytes）

常见的计算机数制（如表 1-4）所示，有 Binary（二进制），Octal（八进制）、Decimal（十进制）、Hexadecimal（十六进制）。除了 Octal（八进制）在 PC 中较少见外，其余在本书中都将经常出现。每一种数制，均用一个基数（radix）来转换。人们常用的十进制，就是用基数 10 来表示数字。

表 1-4　常见的计算机数制

Base 基　数	可　能　的　数　字
Binary（2）	01
Octal（8）	01234567
Decimal（10）	0123456789
Hexadecimal（16）	0123456789ABCDEF

注：十六进制中　A 代表 10　B 代表 11　C 代表 12　D 代表 13　E 代表 14　F 代表 15

1.6.2 Binary Number

计算机中所储存的指令和任何数据均是采用一组二进制的数码表示。所以用 0 和 1 来代表数据与数字是最直接和最适合的。而二进制系统是以 2 为基数，所以只有 0 与 1 两个数字。

1100（Binary）＝12（Decimal）$=1*2^3+1*2^2+0*2^1+0*2^0$

一个数字称为 1bit (0 or 1)。8 个 bit 组成的位串 (加 10101010) 即为一个字节 (1B)，这是所有计算机最基本的存储单位。1 byte 就可代表一个指令、字符或数字。二进制转十进制方法如下：

$$1100\ 0011\ (B) = 1*2^7+1*2^6+0*2^5+0*2^4+0*2^3+0*2^2+1*2^1+1*2^0$$
$$= 195\ (D)$$

习惯上，如果是二进制数字，会在数字最右端加上一个 B，代表是二进制数字，避免与十进制数字产生混淆。

1.6.3 Hexadecimal Number

对人而言，二进制并不易阅读，所以有以 16 为基数的十六进制表示法。前面 10 个用 0 到 9 表示，后 6 个用 A 到 F 表示。十六进制介绍如下：

0011	1100	0101	1001	(B)	(二进制)
3	C	5	9	(H)	(十六进制)

由上例可知，二进制转换为十六进制只须四位一组即可顺利转换。

习惯上，会在十六进制数字最右端加上一个 H，代表是十六进制数字。

十六进制转十进制如下

$1A5h = 1*16^2+10*16^1+5*16^0$

如果你仔细看会发现，我们的计算机系统中，0 代表第一个数字，而不是以 1 为第一个数字，1 就变为第二个数字。

以 1B 表示一数字可发现，在二进制中 11111111 (255) 为所能表达的最大数字，在十六进制中为 0FFh (255)。

二进制加法

```
  0101
 +0011
------
  1000
```

※ 与十进制加法一样，从个位加起，由右往左，超过 1，就进位到左边一位中，依此类推。

十六进制加法

```
  28A70
 +90B6
-------
  31B26
```

※ 与二进制相似，逢 16 便进位。

二进制减法

```
  0101
 -0011
------
  0010
```

※ 与十进制减法一样，若不够减则向前借位。

十六进制减法

```
  28A70
 -90B6
-------
  1F9BA
```

※ 与十进制减法一样，若不够减向前借位。

1.6.4 有符号数与无符号数

站在人的立场上，数字分为无符号数与有符号数。但计算机在处理数字时，仅将其视为一组 0 与 1 的组合，而并不知道数字是有符号数或无符号数。

1.6.4.1 无符号数

若用 1 个 byte 字节代表一个数字时，所有 8 个 bit 均可用来表示一数字，并没有负数，最小值为 0，最大值为 11111111B＝0FFh＝255。仔细观看可发现最大值为 2^8-1。由此可推知，如以两个 bytes 表示一个数字，则最大值为 0FFFFH＝$2^{16}-1$＝65535。

1.6.4.2 有符号数

有符号数的意思是数字分为正和负。以最高位来表示该数目的正负号，称为符号位（sign-bit）。如果该位数字大于或等于 0，则此数字的符号位必为 0；若该位数字小于 0，则符号位便为 1。

也就是说，如果该数字为 1byte，则只有 7 位（从 0 到 6）用来表示数据。如该数字为双字节，则只有 15 位（从 0 到 14）用来表示数据，以此类推。

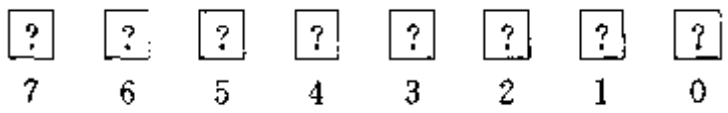
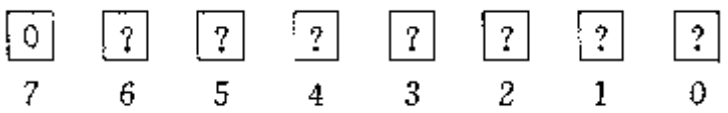
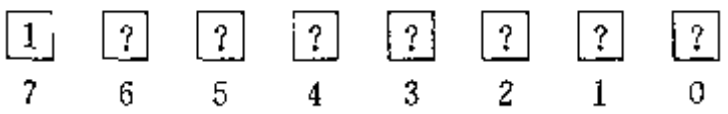


图 1-1　无正负号：所有位均用来表示数字



注：符号位为 0，表示正数



注：符号位为 1，表示负数

图 1-2　有正负号：只有较低的 7 位用来表示数字

在 IBM PC 系统中，负数的表示法是用 2 的补码（2′s complement）来表示。二进制数要表示负数，只要将其相对的正数的二进制数字的所有的位变反的结果加 1 即可得到负数。

变反的意思是将原来为 1 的换成 0，原来为 0 的换成 1。

下面的例子可以让你很清楚地了解负数的转换过程。0 的二进制表示法为 00000000B，所以没有负 0。就算取补码为 11111111B 再加上 1 是 00000000B，还是为 0。

（1）1 的二进制表示法为　00000001B

（2）求反结果　11111110B

（3）再加上 1　11111111B＝0FFH

可知－1 表示为 0FFH，而＋1 表示为 01H。我们可以再发现一个有趣的结果，如果我们将－1 与＋1 相加，则

```
    0000  0001        (1) 表示有进位，为进位位。
  +1111  1111
-------------
 (1) 0000  0000
```

可知 1＋（－1）的结果为零，这与我们所想像的正好相符。

通过上述的讨论，表 1-5 可知 1 byte 可表示由＋127～－128 间的数字（整数）。若为 2 bytes 可代表＋32767～－32768。字节越多则表示范围愈大。

表 1-5

字　长	可代表的整数范围
1	＋127～－128　$(2^7-1 \sim -2^7)$
2	＋32767～－32768　$(2^{15}-1 \sim 2^{15})$
4	＋2147483647～－2147483648　$(2^{31}-1 \sim 2^{31})$

1.6.4.3　内存大小的计算

计算机处理数据是以 byte 为单位，而不是以 bit 为单位。所以要说内存大小，应说成多少 byte。但 byte 还是太小，所以习惯上是以 kilobytes（KB）或 Megabytes（MB）来计算。

$$1\ \text{KB}=2^{10}\ \text{bytes}=1024\ \text{bytes}$$

$$1\ \text{MB}=2^{20}\ \text{bytes}=1048576\ \text{bytes}$$

例如我们会说基本内存为 640KB。主板上的 RAM 有 16MB。

1.6.4.4　BCD 码

BCD 码（Binary Code Decimal）是 IBM-PC 系统另一种数字表示法，这种码特别适合处理十进制数字，包含小数点。

所谓 BCD 码就是用四位表示一个十进制数字。而 BCD 码又可分为压缩式（Packed）和非压缩式（Unpacked）。

1　压缩式 BCD：一连串的四位为一组来表示一个十进制数字。如下：

```
0111 : 1001
  ↑      ↑
  7      9
```

由上例可知一个字节就可存储两个 BCD 数字。

2　非压缩式 BCD：每个字节只以最低四位来表示一个十进制数字；高四位便无定义，就不用它了。如下所示：

```
???? :1001     ???? :0011
  ↑              ↑
  9              3
```

由上例可知存储两个数字就需要两个字节，这比压缩 BCD 码浪费空间。

用 BCD 码代表数字作算术运算时，程序员须作特别的笔记工作，因为计算机均是以二进制码运算，并不知道它是一个 BCD 数字。

1.7　字符存储格式

计算机是用 0 与 1 组成的二进制码存储数字，而其余的英文字母、标点符号也是以二进制码存储。例如 'A'（字符 A）就可以 0100 0001B（41H，1 byte）来存储。计算机常用的编码方式有两种：ASCII 码和 EBCDIC 码。

1.7.1 ASCII

美国标准信息交换码（American Standard Code for Information Interchange 简称 ASCII）几乎可使用在所有微型计算机（包括 IBM-PC 和 IBM PS/2 机器）上。ASCII 码实际上只使用了 7 位，所以有 128 个 ASCII 字符。余下的第 8 位，即最高位，常用作为校验位，意即在传送或接收字符时仍旧使用 8 位，而在解释这个字符意义时，会将最高位忽略。在 IBM-PC 上也使用 extend character set 扩展字符集。值（80H～FFH）表示图形符号、希腊字符、线条、符号、科学符号。

表 1-6

字　符	Binary	Hex
'A'	01000001	41H
'B'	01000010	42H
'C'	01000011	43H
'1'	00110001	31H
'2'	00110010	32H
'3'	00110011	33H

1.7.2 EBCDIC

EBCDIC (Extended Binary Coded Decimal Interchange Code) 码使用 8 位表示一个字符，最多可以表示 $2^8=256$ 个不同的字符。大部分 IBM 计算机均使用这种编码方式来存储文字、数字或符号。连同校验位算在里面，每一字符用 9 位表示。大多数的计算机使用偶校验。

所谓偶校验是将一个字符中所有表示字符内容的位串“1”的数目加起来，若为偶数个，则设校验位为 0，否则为 1。使其保持偶数个为 1，奇校验则须保持奇数个为 1。

在写汇编语言时，所有变量和指令会比高级语言的限制少，但也相对地提高了危险性，许多细节操作都须程序员自己做。

第 2 章　硬件与软件概念

这章解释用汇编设计程序的一般概念。第 1 节复习一些可用的处理器和操作系统以及它们如何协同工作。第 2 节起将开始 CPU 的寄存器组织。如果你不知有那些寄存器可以使用，那大概也不必写汇编语言程序了。就如同不懂九九乘法表要学算术的情况是一样的。第 3、4 节描述系统软件与内存之间的关系，这也是相当重要的基础概念。最后一节摘要描述 MASM 所提供的许多程序设计过程中会用到的工具。

2.1　与汇编语言关系密切的计算机硬件

1. CPU (Central Processing Unit)　这是每台计算机必备的部件，常以其代号代表此计算机名称。常见的有 XT、AT、80286、80386、80486、Pentium。越后面的功能越强大，速度越快。例如 8086（有 16 位的数据总线）就比 8088（只有 8 位的数据总线）快。

16 位数据总线允许你在 8086 处理器使用 EVEN 和 ALIGN 而拥有 WORD 边界的数据，因此可增加数据处理的效率。在 80486 以上的 CPU 大多内置数学浮点运算器 FPU (Floating Point Unit)，这对于 CAD/CAM、3D 等绘图软件的执行速度有很大的帮助。而随着 WINDOWS 与多媒体的流行，大量图形处理对 CPU 速度的要求也越来越高，可见选择适合自己的 CPU 也是很重要的。

2. Memory（RAM、ROM）　RAM (Random Access Memory)，在主机板常看到有 256KB、512KB　SRAM Memory 还有 RAM 插槽上有扩充 RAM，就是我们所常讲的 1MB、4MB、8MB、16MB 的 RAM。ROM (Read Only Memory)，在主机板上常见有 ROM BIOS (如 AMI、AWARD 等)。

3. Serial Port（串行口）　Parallel Port（并行口）　一个 Serial Port（串行口），也称为 Asynchronous (异步) Port。通常在 IBM-PC 做异步传输时，我们使用标准 RS-232 接口。DOS 提供了两个串行口称为 COM1、COM2。它可以连接 MOUSE 或 MODEM。Parallel Port 并行口可同时传送 8 位数据，DOS 提供 3 个 Parallel Port 称为 LPT1、LPT2、LPT3，通常用来连接打印机。

2.2　CPU 寄存器

在 CPU 内部有一些寄存器 (Register)，可用来储存临时数据、内存地址及执行的指令。由于它位于 CPU 内部，可直接接受 CPU 控制，不必通过内存接口按照地址来传送数据，所以具有高速的处理能力。汇编语言能直接存取寄存器，这无形中提高了速度。8086 家族处理器有相同的基本 16 位寄存器集合。

每个处理器也可将某些寄存器视为两个分离的 8 位寄存器。80386/486 有扩展的 32 位寄存器。为维持与其它寄存器的兼容性，80386/486 可以存取寄存器如 16 位或 8 位值。图 2-1 说

明在所有以 8086 为基础的处理器所共有的寄存器，每个寄存器有它自己特殊的使用限制。

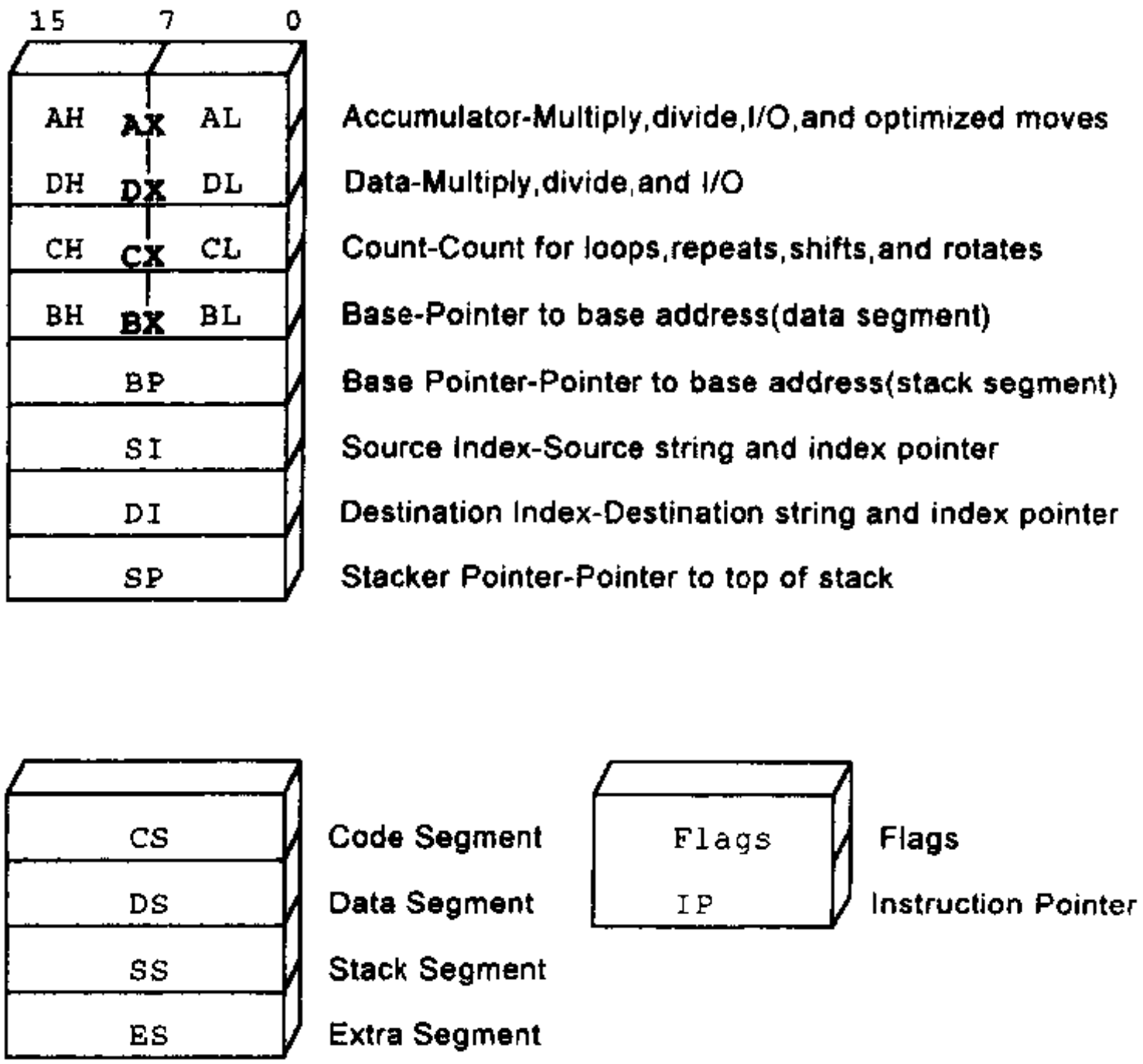


图 2-1　8088、80286 处理器的寄存器

2.2.1　通用寄存器 (General-Purpose Register)、段寄存器 (Segment Register) 和其它寄存器

80386/486 处理器使用相同 8 位和 16 位寄存器（其余 8086 家族使用）。所有寄存器（除了段寄存器总是占据 16 位）都可扩展成 32 位。这些扩展扩充的寄存器的名字开头都使用字母“E”。例如 AX 的 32 位为 EAX。80386/486 还有两个额外的段寄存器，FS 和 GS。图 2-2 说明 80386/486 的扩展寄存器。

2.2.2　段寄存器

有 4 个主要段寄存器（16 位），CS、DS、SS、ES。(80386/486 增加了 FS 和 GS)。我们在设计程序时，可将 FS/GS 视为数据寄存器来存储较少使用到的数据。在执行时，所有地址都是与以上的段寄存器相关联。这些寄存器，他们的段和他们的目的如下：

寄存器和段	目　　的
CS（Code Segment）	存储代码段起始地址。包含处理器指令和它的立即运算符
DS（Data Segment）	存储数据段起始地址。正常是包含由程序配置的数据

（续）

寄存器和段	目　　的
SS（Stake Segment）	存储堆栈段起始地址。包含程序使用的堆栈（如 PUSH、POP、CALL 和 RET）
ES（Extra Segment）	存储额外段起始地址。存取第二数据段。通常是由字符串指令使用
FS，GS	在 80386/486 提供额外段

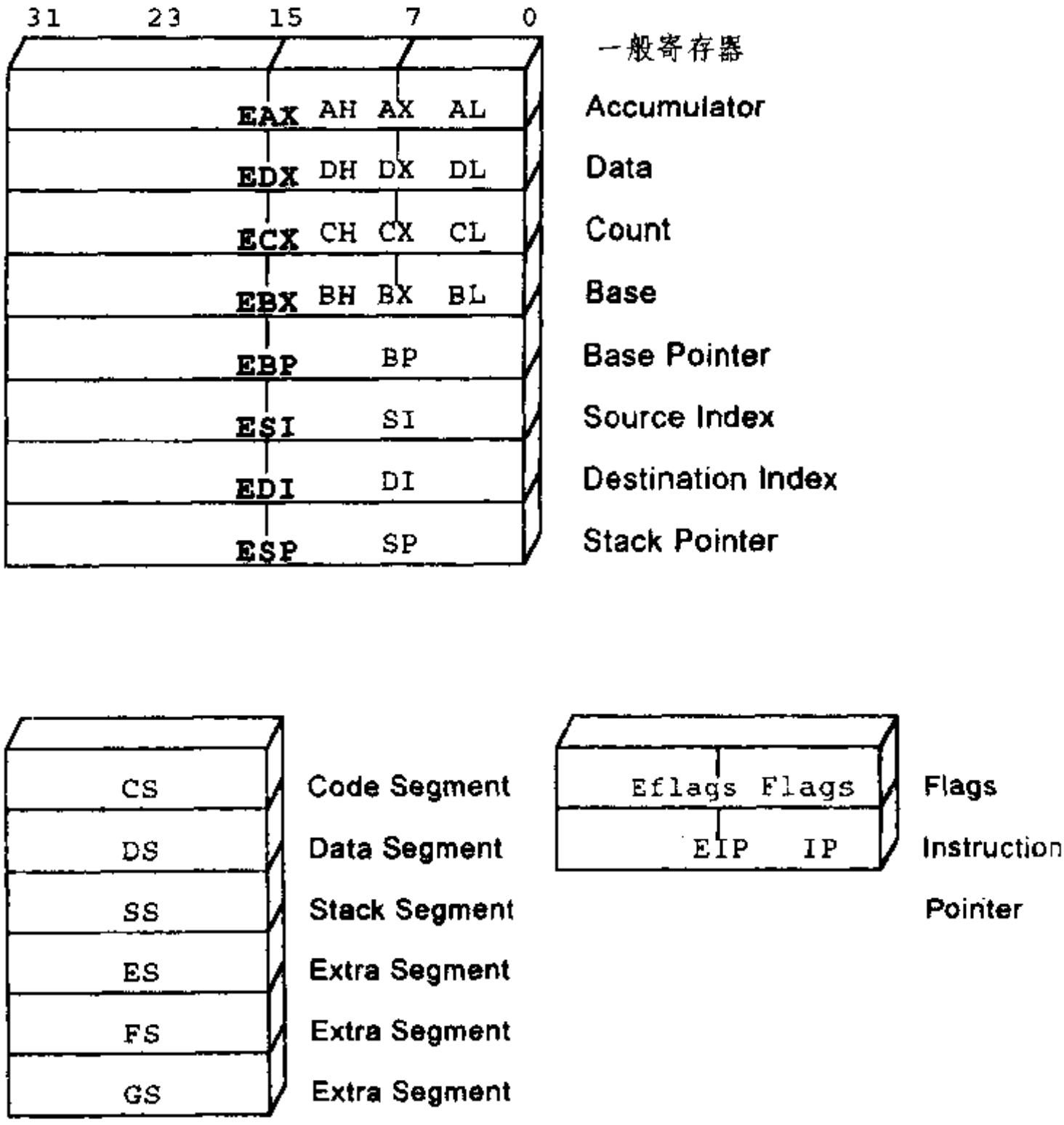


图 2-2　80386/486 扩展寄存器

2.2.3　数据寄存器（又称通用寄存器）

16 位数据寄存器包含 AX、DX、CX、BX、BP、DI 和 SI，通常是做为临时数据储存。因为处理器存取寄存器比存取内存快，你可以使你的程序将经常使用的数据保留在寄存器，使程序执行的快一些。

以 8086 为基础的处理器不能执行内存到内存（memory to memory）的运算。例如，处理器不能将内存中的变量从一个位置拷贝到另一个位置。你必须从内存拷贝到寄存器，然后再

从寄存器拷贝到一个新的内存位置。要相加两个在内存的变量，也是需通过一个寄存器。

每个数据寄存器又有其特定的功能属性，分述如下：

EAX（又可称为累加器；Accumulator）：一般算术运算、输入、输出常使用此寄存器。

EBX（又可称为基址寄存器；Base）：除了算术运算外，还有寻址功能。

ECX（又可称为计数寄存器；Count）：常用作循环执行次数计数或移位指令总移位数。

EDX（又可称为数据寄存器；Data）：较大数值运算时，可与 EAX 配对使用，以 EDX：EAX 代表一数值。

另三个又称变址寄存器：SI、DI、BP。SI 与 DI 寄存器常用来储存数据段内变量的地址。像字符串、数组一些多重数据结构常用到。

ESI（源变址寄存器；Source Index）：常将数据段内源运算符的地址移入 SI 中，处理连续内存中的数据，如字符串。

EDI（目的变址寄存器；Destination Index）：与 DI 搭配进行字符串运算。

EBP（基址指针寄存器；Base Pointer）：可以指向堆栈段或其它数据的间接地址。

一般程序员会遵照上述的寄存器特定功能，以增加程序的可读性，当然也可稍做变换。虽然记住各寄存器的使用规则相当繁琐，会增加初学者负担，但只要多加练习自然会记住各寄存器的功能，相信可使日后程序设计时更加顺利。

2.2.4 特殊寄存器

EIP、ESP 一般程序中是不会改变其值，处理器会自动改变它的值。

EIP（指令指针寄存器；Instuction Pointer）：EIP 寄存器总是储存下一个将要执行的指令在内存中的地址。CS：IP 组合便是一个执行指令的地址。你不可以直接存取或改变这一指令指针，CPU 会自动改变其值。控制程序流程指令（如 calls、jumps、loops、interrupts）会自动改变其值。

ESP（堆栈指针寄存器；Stack Pointer）：ESP 寄存器指向在堆栈段目前的位置。

2.2.5 标志寄存器（EFL）

32 位的寄存器，我们并不将此寄存器当成一个数值来使用，单独使用其中一个独立的位来显示 CPU 的状态或是算数运算的结果，且各有独立的名称。我们并不须自己去分离出每一位，CPU 中有特别指令去帮助我们读取每一位的状态。

标志寄存器的低 16 位控制某些指令的执行和反映目前处理器的状态。在 80386/486 标志寄存器扩展至 32 位。有许多位是未定义的，所以在实模式真正只有 9 个标志，在 80286 保护模式有 11 个标志（包含一个 2 位标志），80386 有 13 个，80486 有 14 个，80386/486 的扩展寄存器有时也称为 "Eflags"。

图 2-3 说明 80386/486 32 位标志寄存器的位。较早期的 8086 家族处理器只使用较低的字。未标明的位是保留给处理器使用，不应该被修改。

状态标志

CPU 在执行算数或逻辑运算时会设置 OF（Overflow Flag）、SF（Sign Flag）、ZF（Zero Flag）、AF（Auxiliary Flag）、CF（Carry Flag）、PF（Parity Flag）这 6 个标志值，也就是会将执行结果反应到相对应的标志中。

（1）进位标志（CF）：执行加法运算时，若有进位会设标志值为 1；否则为 0。执行减法时产生借位也会将标志值设为 1。

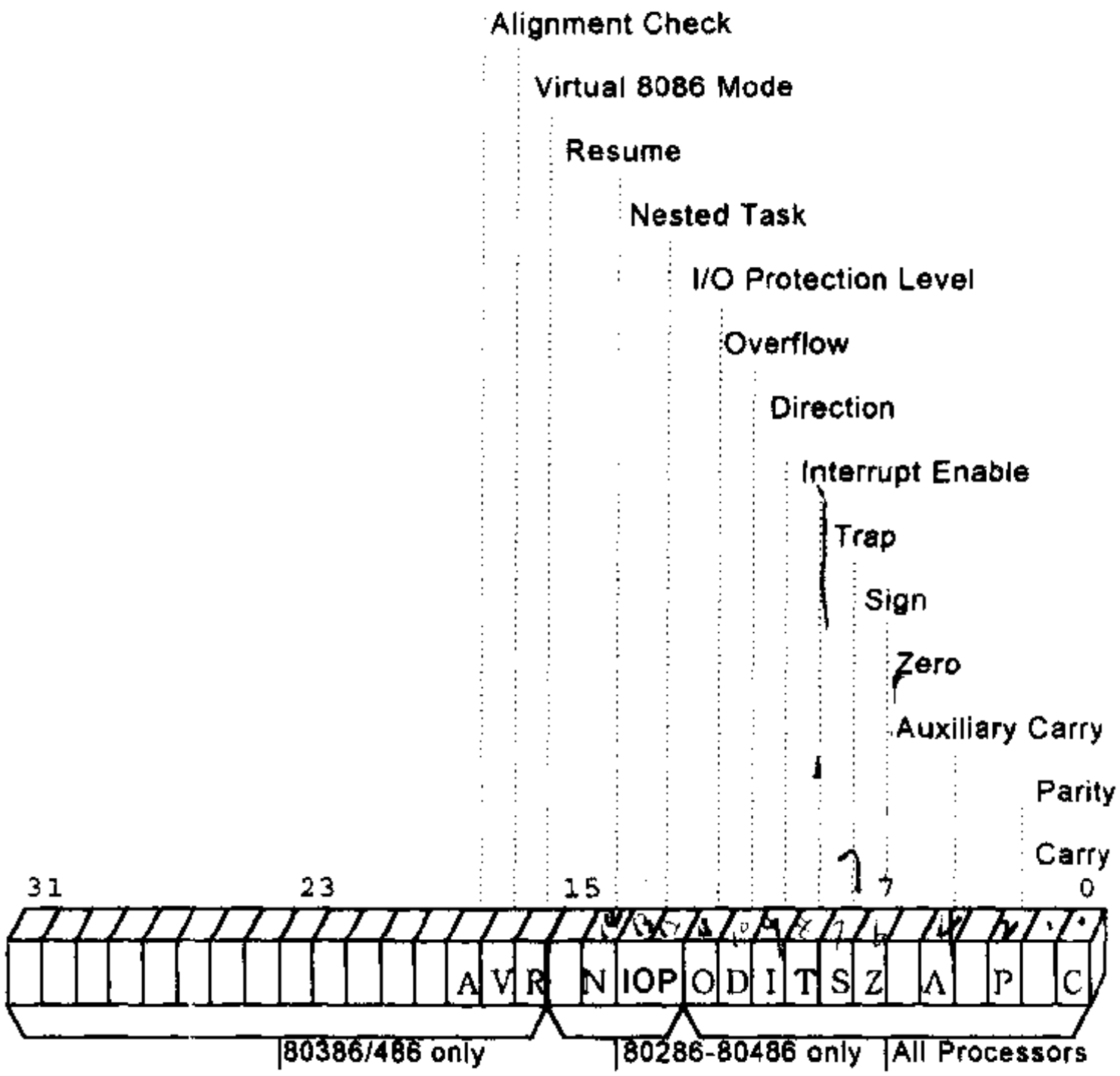


图 2-3　8088～80486 处理器的标志

1=Carry (CY)，0=No Carry (NC)

(2) 溢出标志 (OF)：运算的结果太大或太小，超出目的运算符所能代表的范围时，标志值设为 1；若无则设为 0。

1=Overflow (OV)，0=No Overflow (NV)

(3) 正负号标志 (SF)：算术或逻辑运算的结果若产生一负值，即最高位为 1，则标志值将设为 1；否则标志值设为 0。

1=Negative (NG)，0=Positive (PL)

(4) 零标志 (ZF)：算术或逻辑运算的结果如果产生 0，则标志值将设为 1，否则标志值设为 00。在执行循环或转移指令时，常利用此标志值做判断。

1=Zero (ZR)，0=Not Zero (Zero)

(5) 辅助进位标志 (AF)：运算结果产生从目的运算符的第四位进位或借位时，标志将设为 1；否则为 0。通常是使用在 BCD 码计算时。

1=Aux Carry (AC)，0=No Aux Carry (NA)

(6) 奇偶标志 (PF)：指令运算的结果，所含 "1" 的位总数为偶数个时，标志值设为 1；

若为奇数个时，标志值设为 0。在数据传送时，操作系统会去检查数据的完整性是否无误。

控制标志

控制标志共有 3 个，DF（Direction Flag）、IF（Interrupt Flag）、TF（Trap Flag）。

（1）方向标志（DF）：主要用在字符串指令。STD 指令会设 DF＝1，由高地址往低地址处理字符串，CLD 指令会设 DF＝0，CPU 将从低地址往高地址处理字符串。

1＝UP（UP），0＝DOWN（DN）

（2）中断标志（IF）：CPU 通过此标志值判断是否可接受外部中断。外部中断可由硬件设备如键盘、驱动器、系统计时器所发出。

1＝Enable（EI），0＝Disabled（DI）

（3）单步标志（TF）：如果标志被设为 1，CPU 在每个指令之后都会产生一个单步的中断。一般是在调试（Debugging）程序使用去一次执行程序的一个指令。

虽然所有的标志都各有目的，但大部分的程序只需要 carry、zero、sign 和 direction 标志。

2.3 系统软件与内存

1. PC 内存结构　PC 可以处理 20 位地址(2^{20}＝1MB)的内存，这些内存分别储存于 RAM 和 ROM。RAM 内存由 00000H 到 BFFFFh，ROM 内存由 C0000h 到 FFFFFh。80386 以上的 PC 可以处理 32 位（2^{32}＝4GB）的内存地址。

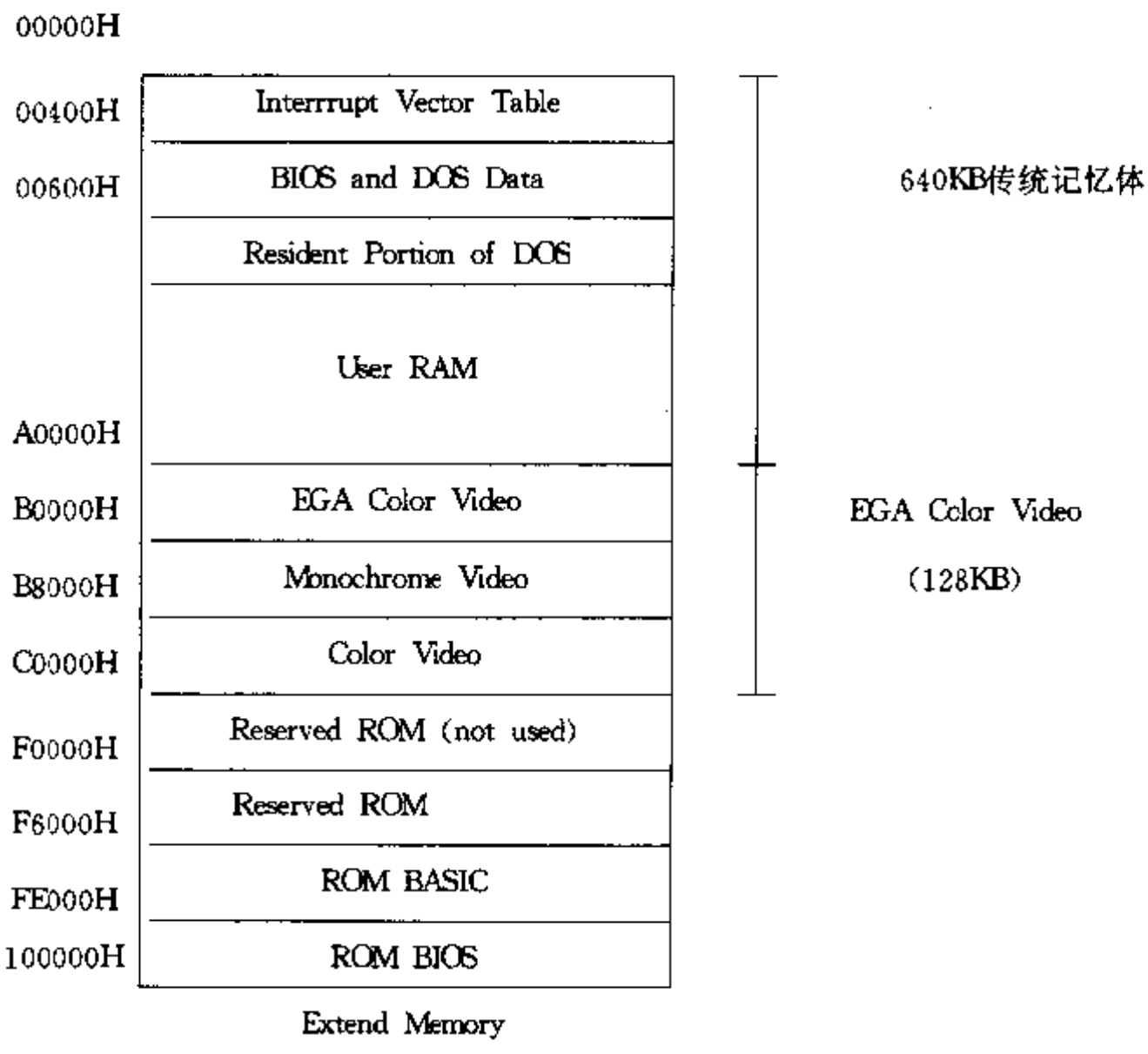


图 2-4　640KB 基本内存

中断向量表：在内存最低地址 0000H 到 003FFH（共 400H，1024 bytes）中，存放有一个中断向量表。每 4 个字节存放一个中断服务程序的中断地址，所以共有 256 个中断地址，CPU 即根据这些地址去处理硬件与软件中断。当一个外部装置或程序要求 CPU 中断服务时，CPU 便通过表中相对应的地址去找到中断服务程序的起始地址，去执行中断服务程序。

2. BIOS 和 DOS 数据　从 00400H 到 005FFH（共 200H，512 bytes），是由 DOS 所使用。

（1）键盘缓冲区　所有按键均会储存在缓冲区中，直到它们被处理。

（2）键盘状态标志　所记录目前键盘使用的情况，如许多特殊键（Ctrl、Shift）是否有被按，大小写（Caps Lock）键、数字（Num Lock）键指示灯是否亮着，均可由此查出。

（3）打印机、串行口的地址。

（4）系统配置的描述　记录内存的大小/驱动器数、屏幕的类型（彩色或单色）。

用户可使用的内存地址：从 00600H 地址起是 DOS 常驻部分，由于 DOS 版本不同，所占空间也会不同，其余至 1MB 均是真正可以执行程序的空间。

3. 屏幕显示内存　屏幕显示方式是采内存对映方式（Memory-Mapped）的，当 DOS 要写一个字符至屏幕，会调用 ROM BIOS，将字符直接写至显示内存中，显示内存（128K）由 A0000H 到 BFFFFH。单色显示器（Monochrome）只使用 4K（B0000H 到 B7FFFH）。

CGA (Color Graphics Adapter) 使用 16K (B8000H 到 BBFFFH)。EGA (Enhanced Graphics Adapter) wget 128K (A0000Hgc BFFFFH)。程序可以直接将字符写到相对应的内存地址以加快速度。不过不同显示卡有不同的相对应显示地址，必须先检查是哪种卡。

4. ROM 区　C0000H 到 FFFFFH 是保留给 ROM 使用，F0000H 到 FFFFFH 是 ROM BIOS 区，包含有供 DOS 使用的 Input、Output 的低级子程序，还有 BASIC 函数。

2.4 段寻址

段寻址是内部技巧（结合一个段地址和一个偏移地址去形成一个完整的内存地址）。一个地址的两个部分可被表示成：

Segment：offset

段地址部分总是包含一个 16 位值。偏移地址部分在 16 位模式时是一个 16 位值，32 位模式是一个 32 位值。

对于实模式（real mode），段值是一个实际的地址（需加偏移地址再计算）。结合段地址和偏移地址可以建立一个 20 位的实际地址。虽然 20 位的地址可以寻址到 1MB 的内存，但在国际标准结构（IBM 及兼容）计算机的 BIOS 和操作系统使用这部分的内存，剩下可用的内存则留给程序使用。

段运算

直接在实模式设计程序时，计算段和偏移地址称为段运算（segment arithmetic）。

80x86 的 CPU 是用两段式寻址方式来存取内存的值。所谓两段式寻址，是将 16 位的段寄存器和 16 位的偏移地址组合以存取 20 位的线性地址。实际上，段是选择一个 64KB 的内存段，偏移是选择在此段的位。其存取的方式如下：

（1）处理器会将段地址左移四位，产生一个 20 位的地址。最后边空出的四个位置则用四个 0 补上。这样的运算就等于将段地址乘上 16。

（2）处理器会将 20 位的段地址加上 16 位的偏移地址。偏移地址是不需被移位的。

（3）处理器使用最终的 20 位地址，称为“实际地址(physical address)”，去存取在这 1MB 地址空间内的一个真正位置。

由图 2-5 可知地址表示方式由 Segment：Offset 组成，CPU 计算实际地址时会将 Segment 值往左移 4 位即乘上 16 以求得真正地址。

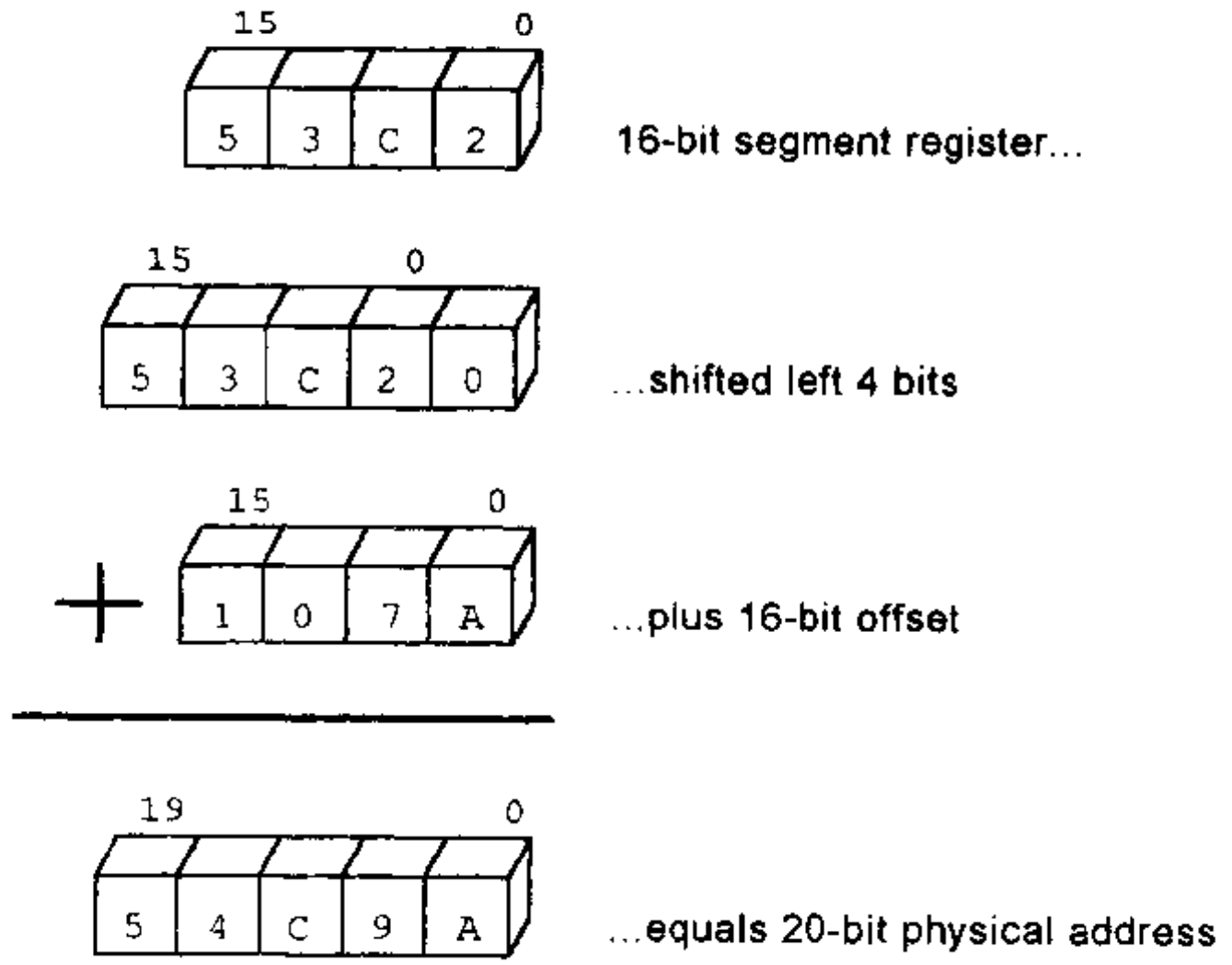


图 2-5　计算实际地址

一个 20 位实际地址可以由 segment：offset 地址指明。例如，地址 0000：F800，0F00：0800，和 0F80：0000 都是指向相同的实际地址 0F800。

2.5　预处理符号

编译程序包含许多预处理符号 predefined symbols（也称为 predefinedequates）。你可以在程序任何地方使用这些符号去表示其实际值。例如

@FileName 表示目前文件的名称。如果目前的源文件（Source file）是 TASK. ASM，@FileName 的值是 TASK。如果在命令行 ML 之后使用/Cp，则区分大小写。

有关段信息的预处理符号如下所示：

符　号	描　述
@code	Returns the name of the code segment.
@CodeSize	Returns an integer representing the default code distance.

（续）

符　号	描　述
@CurSeg	Returns the name of the current segment.
@data	Expands to DGROUP.
@DataSize	Returns an integer representing the default data distance.
@fardata	Returns the name of the segment defined by the. FARDATA directive.
@fardata?	Returns the name of the segment defined by the . FARDATA? directive.
@Model	Returns the selected memory model.
@stack	Expands to DGROUP for near stacks or STACK for far stacks.
@WordSize	Provides the size attribute of the current segment.

有关环境信息的预处理符号如下所示：

符　号	描　述
@Cpu	Contains a bit mask specifying the processor mode.
@Environ	Returns values of environment variables during assembly.
@interface	Contains information about the language parameters.
@Version	Represents the text equivalent of the MASM version number. In MASM 6.1, this eapands to 610.

有关时间和日期信息的预处理符号如下所示：

符　号	描　述
@Data	Supplies the current system date during assembly.
@Time	Supplies the current system time during assembly.

有关文件信息的预处理符号如下所示：

符　号	描　述
@FileCur	Names the current file (base and suffix) .
@FileName	Names the base name of the main file being assembled as it appears on the command line.
@Line	Gives the source line number in the current file.

有关宏字符串处理信息的预处理符号如下所示：

符　号	描　述
CatStr	Returns concatenation of two strinfs.
@inStr	Returns the starting position of a string within another string.
@SizeStr	Returns the length of a given string.
@SubStr	Returns substring from a giving string.

2.6 条件伪指令

MASM 6.1 提供了条件编译伪指令和条件错误伪指令。条件编译伪指令可让你测试一个指定的条件，如果条件为真则编译一组指令。条件错误伪指令允许你测试一个指定的条件，如果条件为真，则产生一组编译错误信息。注意，条件编译伪指令只能测试编译时的条件，而不是执行时的条件。你也可以测试一个在编译期间可以计算的运算式。

2.6.1 条件编译伪指令

IF 和 ENDIF 伪指令包含这些条件叙述。选择指令 ELSEIF 和 ELSE 接在 IF 伪指令之后。下面的叙述说明 IF 伪指令的语法。其它的条件编译伪指令遵循同样的格式。

```
IF    表达式 1
      指令组
[ELSEIF 表达式 2
      指令组]
[ELSE
      指令组]
ENDIF
```

在 IF 块内的指令组可以是任何有效的指令，包括其它的条件编译伪指令。只有相关的条件为真，MASM 才会编译 IF 伪指令之后的指令组。如果条件为假且块内包含有 ELSEIF 伪指令，编译程序会检查相关的条件是否为真。如果为真，则编译程序会编译 ELSEIF 伪指令之后的指令组。如果 IF 或 ELSE IF 条件没有被满足，编译程序只会处理 ELSE 伪指令之后的指令组。

例如，你可以编译在程序中定义的某个标记之后的一列程序码。

```
IFDEF     buffer
buff      BYTE     buffer     DUP (?)
ENDIF
```

编译程序只有当前面已定义了 buffer 才会配置 buff。

MASM 6.1 提供 IF1、IF2、ELSEIF1 和 ELSEIF2 伪指令去允许只编译第一阶段或第二阶段。不过要使用这些伪指令你必须将 5.1 的兼容性设置（在命令行使用/Zm 或 OPTION M510）置为 enable 或设置为 OPTION SETIF2：TRUE。

下面摘要列出条件编译伪指令：

Directive	Grants Assembly IF
IF expression	expression is true (nonzero)
IFE expression	expression is false (zero)
IFDEF name	name has been previously defined
IFNDEF name	name has not been previnsly defined
IFB argument*	argument is blank

（续）

Directive	Grants Assembly IF
IFNB argument*	argument is not blank
IFIDN [I] arg1, arg2*	arg1 equal arg2
IFDIF [I] arg1, arg2*	arg1 does not equal arg2 The optional I suffix (IFIDNI and IFDIFI) makes comparisons insensitive in case.

注：*只在宏中使用。

2.6.2 条件错误伪指令

你可以使用条件错误伪指令去调试程序和检查编译过程的错误。通过在程序中插入一个错误伪指令，你可以在编译时测试此位置的条件，也可使用在宏中。

和其它严重的错误一样，由错误伪指令产生的错误可以造成编译程序传回一个非 0 的退出码。如果 MASM 在编译时遇到一个严重的错误，它是不会产生目标模块（object module）的。

例如，如果程序没有定义一个给定的标记，.ERRNDEF 伪指令会产生一个错误。在下列的例子中，.ERRNDEF 会确定一个称为 publevel 的标记是否真正存在。

```
.ERRNDEF publevel
IF       publevel LE 2
PUBLIC   val, var2
ELSE
PUBLIC   var1, var2, var3
ENDIF
```

EQ、NE、LT、LE、GT、GE 是关系运算符。下面摘要列出条件错误伪指令：

Directive	Generates and Error
.ERR	Unconditionally where it occurs in the source file. Usually placed within a conditional-assembly blocks.
.ERRE expression	If expression is false (zero) .
.ERRNZ expression	If expression is true (nonzero) .
.ERRDEF name	If name has been defined.
.ERRNDEF name	If name has not been defined
.ERRB argument*	If argument is blank
.ERRNB argument*	If argument is not blank
.ERRIDN [I] arg1, arg2*	If arg1 equals arg2.
.ERRDIF [I] arg1, arg2*	If arg1 does not equal arg2. The optional I suffix (.ERRIDNI and .ERRDIFI) makes comparisons insensitive to case.

注：*仅用于宏中。

第 3 章　汇编语言程序

本章将开始学习编写一个汇编语言源程序(Source Program)并将它编译为一个可执行程序。在这章中将解释如何去开始、结束、组织段。MASM 6.11 提供了改进的 STARTUP 伪指令去帮助我们启动与结束程序。最后将介绍在汇编语言中最常使用也最重要的 MOV、PUSH、POP 指令，这可以说是学习汇编的基本功。

3.1　编译流程

图 3-1 是汇编语言程序编译、连结、执行的过程。

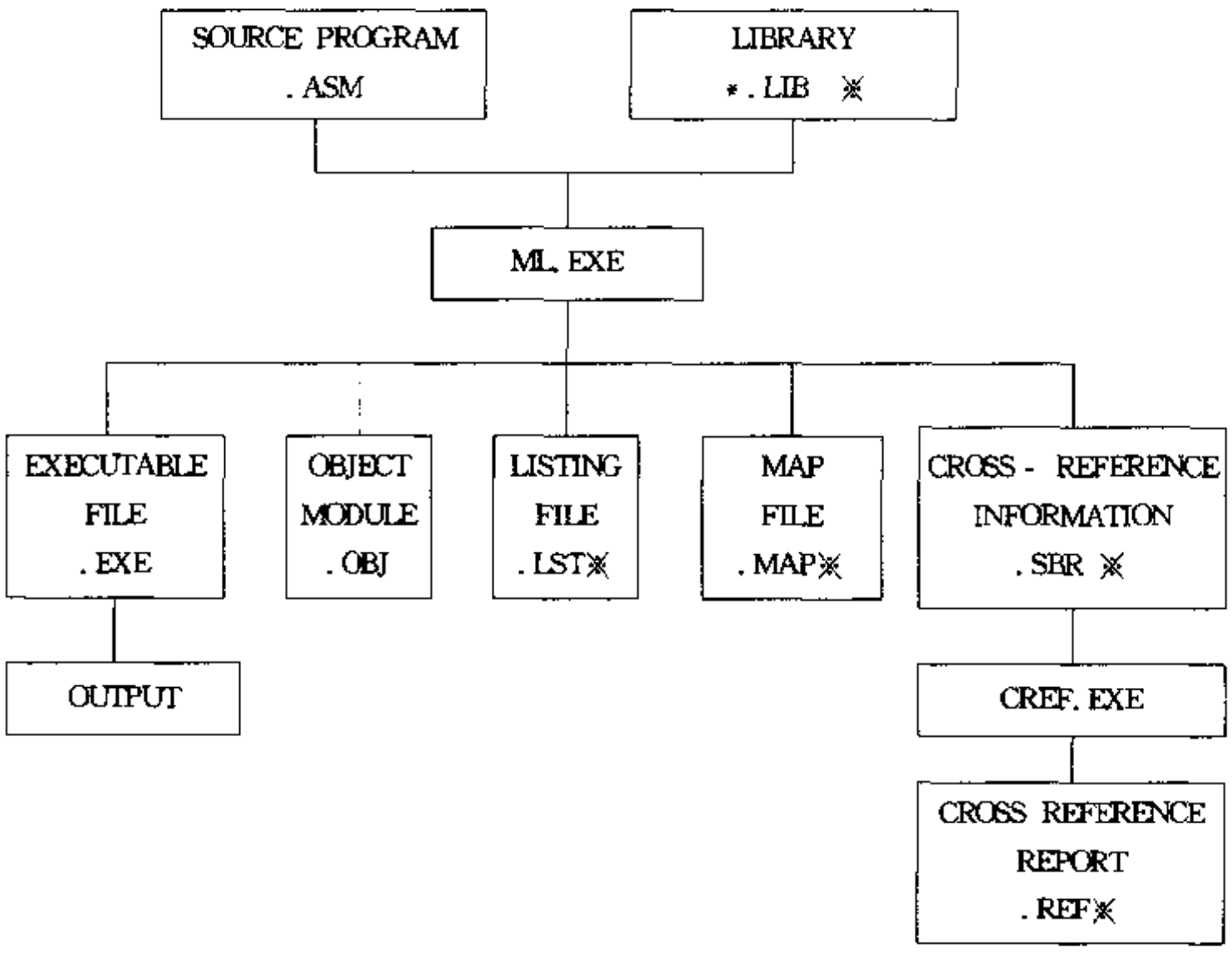


图 3-1　编译、连结、执行的过程

注：※选择性文件

首先你可以用文本编辑器，如编辑和存储一个源程序文件（Source Program）。注意：请以 ASM 为扩展名存储文件，而且文件内容要以 ASCII 格式存储文件。仔细观察应该发现上述整个过程与高级语言 C、Pascal、Basic 相当类似。

从 6.0 版后，Microsoft 将 Assembler 和 Linker 组合成一个 ML 程序。以前需要两个步骤（编译、连结）才能生成一个 EXE 文件，现在只要一个步骤就可以。注意，程序扩展名一定要是 ASM，而且执行时一定要输入扩展名。应用实例如下：

ML　DISPLAY. ASM

执行后将产生 DISPLAY. OBJ 和 DISPLAY. EXE。因为缺省只会产生 OBJ 和 EXE 文件，若要产生 LST、MAP、SBR 文件可在 ML 之后加/Fl、/Fm、/Fr 等参数即可。各参数之间一定要用空格分开，因为直接加在 ML 之后，所以又称 Precedence options，应用实例如下：

ML/Fl　/Fm　/Fr　DISPLAY. ASM

表 3-1 是各参数使用的意义。全部均可选择性的使用。

表 3-1　参数使用说明

参　数	说　明
/Fl	建立一个 LST 文件
/Fm	建立一个 MAP 文件
/Fr	建立一个 SBR 文件

你也可以只加/c 参数，这将使 ML 只编译产生 OBJ 文件，而不会连结产生 EXE 文件。记住是小写的 c。应用实例如下：

ML/c DISPLAY. ASM

3.2　执行程序

一旦你已有了可执行文件（EXE 文件），你可以用两种方式执行。

1. 你可以在 DOS 命令行键入 DISPLAY (Enter)，假设 DISPLAY. EXE 在当前目录下或 PATH 指定的路径下。若不是上述情况，你也可直接指定路径。假设 DISPLAY. EXE 在\WORK\PROGRAMS 的子目录，你可以键入：

\WORK\PROGRAMS\DISPLAY

或在 PWB 下执行。

2. 第二种方式是在 Debugger 下执行。如 DOS 的 Debug 或 CodeView (Microsoft)。它的好处是可以很容易去调试程序，查看或立即修改寄存器的值。

3.3　程序结构

编写汇编语言程序有下列几个主要的原则：

（1）第一行只能包含一个叙述（statement）。

（2）一个指令可以出现在任何地方。

（3）你可以使用大写或小写，因为编译程序（Assembler）是不会区分大小写，无论大小写都是一视同仁。只有在表示字符串时（如 "Hello"，或 'Hello'），才会区分大小写。

3.3.1　组织段

在汇编语言中了解段（Segment）是程序设计的一个重要部分。在以 8086 为基础的处理器中，段这个名词有两个意义：

（1）一个分离的内存块（Memory Block）称为一物理段（Physical Segment），这个物理段的大小在 16 位处理器为 64KB，32 位处理器为 4GB。

（2）一个可变化大小的内存块称为逻辑段（Logical Segment），它是由程序的代码（code）或数据（data）所使用。

在简化段这节中将解释如何去开始、结束、组织段。也会解释如何使用简化段伪指令（directive）去存取远程（far）的数据及程序。

在附录C中描述如何排列（order）、组合（Combine）、区分段；使用SEGMENT伪指令去定义完整段（full segment）；如何建立一个段组（group），这样你可以使用向一个段地址去存取所有数据。

3.3.2 物理内存段（Physical Memory Segment）

一个物理段可以开始于一个能被16整除的内存地址（包含0）。Intel称这样的地址为"paragraphs"。你可以很容易地认出一个paragraph的地址，因为它的16进制地址总是以0结束。如A0000H，2EB90H。8086/286的处理器允许段有64KB大小。80386/486在实模式下还是坚持64KB的限制。然而在保护模式（ProtectedMode）下，使用32位寄存器可以寻址到4GB。

段的结构对许多汇编语言程序员会造成许多困难。对于小模式（Small Mode）而言，这个限制却不重要。因为每一个code和data都各自使用小于64K的独立段。在一个段中一个简单的offset就可配置一个指令或变量。

然而在较大的程序中，必须争论段内存区段的问题。如果数据使用两个或较多的段，必需指定段（segment）和偏移（offset）地址去存取一个变量。

3.3.3 逻辑段（Logical Segment）

逻辑段包含程序的三个部分——code、data、stack。MASM会替你组织这三个部分使他们使用内存的实际段。段寄存器CS、DS和SS包含的逻辑段存在于实际内存段的地址。

你可以用两种方法定义段——使用简化段伪指令和完整段定义。你可以在相同的程序中使用两种段定义。简化段伪指令隐藏了许多段定义的细节和假设，与高级语言有相同的使用习惯。简化段伪指令产生必须的程序码，指定段属性和安排段顺序。

完整段定义需要较复杂语法，但是提供较完整的控制。如果你使用完整段定义，你必须自己写程序去处理许多原本由简化段伪指令自动执行的工作。

3.4 使用简化段伪指令

编译程序提供一些伪指令（directive），使你能控制一个程序进行编译与控制列表的进行（指运用在LST文件时），所以又称之为虚拟运算指令（pseudo code），它只对编译程序有作用，而不会产生任何机器码。

使用简化段伪指令　要将MASM程序结构化，需要在你的程序中使用许多伪指令（directive）去指定标准的names、alignment和属性。利用这些伪指令定义的段，将它和高级语言连结较容易。

这些简化段伪指令是.MODEL、.CODE、.CONST、.DATA、.DATA?、FARDATA,、FARDATA?、.STACK、.STARTUP和.EXIT。

MASM程序包含许多由段组成的模块。每一个用MASM写成的程序只能有一个主模块（一个程序开始执行的地方）。这个主模块（主程序）可以包含用简化段伪指令定义的code、da-

ta、stack 段。任何其它额外的模块（子程序）应该只能包含 data 和 code 段。然而每个模块（使用简化段）必须要以 .MODEL 伪指令开头。

我们将详细介绍汇编语言的格式，因为这是编写汇编语言最重要的基础，如果没有真正完全弄清楚，则将给以后的学习增加困难，所以我们将花极大的篇幅来详细介绍。

接下来我们开始介绍基本的汇编语言程序结构。一个程序可分为许多段，在 5.0 版以后就提供了简化段的指令。程序分成许多段的目的是希望将数据与程序放在各自的段里以利于管理，不致混乱，而且结构也较好。

下面的实例显示了使用简化段伪指令的主模块（程序）结构。程序是使用预设的处理器（8086）和缺省的堆栈距离（stack distance）；NEARSTACK。其它额外的模块应该只能使用 .MODEL、.CODE 和 .DATA 伪指令与 END 指令。本书的程序将大部分采用下面的格式。

```
        PAGE 66，80      ；程序的目的是要在屏幕上显示一行信息
        TITLE DISPLAY A MESSAGE
.MODEL SMALL
.386
.DOSSEG
.STACK 1024
.DATA
        message BYTE'Hello，how are you?'，0DH，0AH
        message-length EQU $-message
.CODE
            PAGE
    main    PROC
            .STARTUP
            MOV AH，40H
            MOV BX，0001H
            MOV CX，message-length
            LEA DX，message
            INT 21H
            .EXIT
        main    ENDP
    END
```

1. PAGE　PAGE 是一条伪指令 (DIRECTIVE)，指明在 LST 文件每一页中所要打印的行数及每一行最多的字符数。只运用在编译时产生列表的程序控制。并不是一个真正的指令，所以也称为伪指令，它并不会产生机器码。其中 66 与 80 是缺省值。66 代表每一页最多的行数，(可使用的范围是 10～225)。80 代表每行的字符数，(可使用的范围是 60～132)。假设行数设置为 66，表示编译程序打印 66 行后会自动跳页到下一页的开头位置。若要在程序列表中强迫换页，可在程序任何地方加上 PAGE 指令，不需要运算符。

2. TITLE　若要在每一页打印时在开头加上标题，可使用 TITLE 指令。

3. MODLE　你可以将 .MODEL 放在任何简化段指令之前，语法是：

.MODEL　内存模式［，模块选择］

此内存模式参数是应立即出现在 .MODEL 指令后而且是必须的。模式选择与模式参数中间用（,）隔开。当然你也可选择不写，而直接在 ML 之后加上相对应的参数。

表 3-2 是内存模式栏和模块选择栏所指定的语言和堆栈距离。

表 3-2

栏	说　明
内存模式	TINY、SMALL、COMPACT、MEDIUM，LARGE、HUGE 或 FLAT。要决定 Code 和 Data 段的大小，此栏是必须的
语言	C、BASIC、FORTRAN、PASCAL、SYSCALL 或 STDCALL，设置所要连结调用的语言（用于子程序居多）和公用符号

4. 堆栈距离　NEARSTACK 或 FARSTACK。如果指定 NEARSTACK 则会将堆栈段和一个单一实际数据段结合，SS 寄存器会假设和 DS 值相等。若指定 FARSTACK 则不会将数据段和堆栈段结合在一起，也就是 SS 值不会与 DS 相等。

你可以只使用一个栏或使用多个，但内存模式栏是一定不能少的。下面是应用实例。

```
.model small                    ; Small memory model
.model large, c, farstack       ; Large memory model,
                                ; C conventions
                                ; separate stack
.model medium, pascal           ; Medium memory model
                                ; Pascal conventions
                                ; near stack (default)
```

3.5　定义内存模式

MASM 为高级语言使用提供了标准的内存模式——tiny、small、medium、compact、large、huge 和 flat。你可以在 .MODEL 面后指定上述名称。有一个例外的情况是 flat 模式，规定需要使用 80386/486 指令。其余的情况下你选择的内存模式并不限制你使用指令的种类。然而内存模式会控制段(Segment)和决定是否 data 和 code 为 near 或 far。所有使用均在表 3-3 中。

当所写的汇编语言程序模块（子程序）要和高级语言进行连结时，高级语言与汇编程序所用的内存模式必须相同。建议使用 small 模式，因为此模式的 code 与 data 均采用 near 属性存取（即在 64KB 之内），比 Far 属性的远程存取（可存取 64KB 之外）效率更高。

表 3-3　内存模式属性

Memory Model	Default Code	Default Data	Operating System	Data and Code Combined
Tiny	Near	Near	MS-DOS	Yes
Small	Near	Near	MS-DOS, Windows	No

（续）

Memory Model	Default Code	Default Data	Operating System	Data and Code Combined
Medium	Far	Near	MS-DOS，Windows	No
Compact	Near	Far	MS-DOS，Windows	No
Large	Far	Far	MS-DOS，Windows	No
Huge	Far	Far	MS-DOS，Windows	No
Flat	Near	Near	Windows NT	Yes

3.5.1 Small、Medium、Compact、Large 和 Huge 模式

在许多语言中，传统上我们所认识的内存模式有 small、medium、compact、large 和 huge。small 模式提供一个数据（data）段和代码（code）段。所有的 code 和 data 缺省值都是 near 近程存取。也就是不能超过 64KB。large 模式提供多重的数据（data）段和代码（code）段。均是采用 far 远程存取，也就是 code 和 data 段大小均可超过 64KB。每一种模式都可再定义成其它属性。例如在 small 模式有 large 数据项目，或在 large 模式的程序内的过程（procedure）有 near 属性。

内存模式	程序范围	数据范围
Small	64KB	64KB
Medium	1MB	64KB
Compact	64KB	1MB
Large	1MB	1MB
Huge	1MB	1MB

注：在实模式下仍只限于 1MB

3.5.2 Tiny 模式

Tiny 模式只能使用于 MS-DOS 下。Tiny 模式在单一的段中放置所有的 data 和 code。因此所有程序的大小不能超过 64KB。缺省值是 near 属性的 code 段和静态数据项目（static data items），而且不可以更改这些缺省值。然而你可以使用 DOS 内存管理服务程序在 run time 时动态地管理 Far data。Far data 是指数据存取时需参考到 segment 和 offset。而 near 属性只用到 offset，因为所有数据都在同一段中，而一个段的范围是 64KB。所以 far 属性的数据可以超过 64KB 大小。）

Tiny 模式的程序产生 MS-DOS 的 COM 文件。指定了 .model tiny 会自动向边结程序送 /TINY 参数。所以你无需在 ML 之后加上/AT 参数去生成 COM 文件。

3.5.3 Flat 模式

Flat 模式在 32 位的操作系统下是没有段的设置，因为它和 tiny 模式一样，所用的 .code 和 data 都在同一段中，而 tiny 的段是 16 位，所以大小不超过 64KB2^{16}，而 Flat 模式的段是 32 位，大小可至 2^{32}（4GB）。

写一个 Flat 模式程序需要在写 .Model Flat 之前指定 .386 和 .486。所有的 code 和 data

(包含系统源)是在一个单一的 32 位段中,系统会自动在载入 RAM 时指定段寄存器(segmentregisters)。只有在你必须要混合用 16 位和 32 位段时才需要修改。CS、DS、ES 和 SS 都在 Supergroup FLAT 中。

3.6 设置 Stack Distance

此 NEARSTACK 会将 STACK 段放在 DGROUP 的组中,也就是和 DATA 段相同的组,而 .STARTUP 会调整 SS:SP 使 (SS=DS)。在这种情况下(即可使用 NEARSTACK,缺省值),若没有使用 .STARTUP,则很可能使程序不能执行。此 FARSTACK 设置将使 STACK 有它自己的段,也就是 (SS≠DS)。缺省的设置 STACK 形式是 NEARSTACK,对大部分的程序是一个很方便的设置。因为这种情况下你可使用 DS 去存取 STACK 内的数据或用 SS 去存取 NEAR 属性的数据,因为是同一个段。

FARSTACK 是使用在特殊情况下,如常驻程序或动态连结库程序。

3.7 指定处理器(Processor)和协处理器(Coprocessor)

MASM 提供一组指令去选择 processor 和 coprocessor。一旦你选择一个 processor,你只能使用此 processor 的指令集 (instruction set)。而缺省值是 8086 处理器。如果你总是要你的程序执行在 8086 上,那么你不需要加入任何处理器指令 (processor directive)。使用 .186、.286、.386 或 .486 指令时,可选择不同的处理器及其所提供的额外指令。使用 .8087 (缺省值)、.287 或 .387 可选择所要使用的数学协处理器指令集。而使用 .NO87 可不编译所有协处理器指令。

注意 .486 可使所有协处理器指令被编译,因为 80486 或以上的处理器内置有完整的 coprocessors 的寄存器和指令。每种处理器指令都各自相对于一个相关联的协处理器指令。

3.8 建立 Stack

Stack 是当一个子程序被调用时,作为 push 或 pop 寄存器值和储存返回原地址的内存段。所以 Stack 通常是储存暂时和段变量。

如果你的主程序是用高级语言写的,那么高级语言会自动替你处理建立 STACK 的细节。.Stack 指令只用于当你的主程序是用汇编语言写成的情况下。

.Stack 指令会建立一个堆栈段 (stack segment)。缺省时,编译程序会分配 1KB (1024 bytes) 的内存给堆栈使用。这个大小已足够大部分的小程序使用。

```
.STACK      2048      ; 使用 2KB (2048 bytes) 的堆栈区
```

3.9 建立数据(Data)段

程序可以包含 near 与 far 数据。一般来说你应该在 near 数据段设置重要的和经常使用的数据。因为在这个区段内的数据存取较快。但是这个区段可能是比较拥挤的,因为 16 位操作

系统所有的 near 数据和模块不能超过 64KB 大小。因此你可以在 far 数据段中设置不经常使用或特别大的数据。

.DATA、.DATA?、.CONST、.FARDATA 和 FARDATA? 指令可用来建立数据段。

这五条指令也可避免指令出现在数据段中（如果 CS 值错误）。

3.9.1 Near Data Segment

.DATA 建立一个 near 数据段，这个段包含经常使用的数据。在 MS-DOS 下占 64KB 或在 Windows NT 的 Flat 模式下占 512MB 大小，如果它是放在 DGROUP 中只能有 64KB 大小。

当你使用 .MODEL，编译程序会自动在你的 NEAR 数据段定义 DGROUP。从 DGROUP 段中存取 NEAR 数据，正常是直接通过 DS 或 SS。你也可以在 DGROUP 中定义 .DATA? 和 .CONST 段。

虽然这些段最终都会结合在一起（含 STACK）且使用时就如同数据段一样；但使用 .DATA? 和 .CONST 可加强它与 Microsoft 高级语言的兼容性。在 Microsoft 语言中，.CONST 用来定义常量 (constant)；如必须被储存在内存的字符串或浮点数。.DATA? 段用来储存未初始化的变量。

3.9.2 Far Data Segment

Compact、large 和 huge 模式使用 far 数据地址。而在这些模式中，你还是可以使用 DATA、DATA? 和 CONST 去建立数据段。这些伪指令并不会因为内存模式的不同而有不同的使用结果。他们总是促成段在缺省的 DGROUP 数据段，所有的大小限制在 64KB 内。

当你在 small 或 medium 内存模式中使用 FARDATA 或 FARDATA?，编译程序会各自建立 far 数据段 FAR _DATA 和 FAR _BSS；你可以这样存取变量。

```
mov   AX,   SEG farvar2
mov   DS,   AX
```

3.10 建立 Code 段

无论你写一个主模块或一个被其它模块调用的子模块，你都可以有 near 和 far 代码 (code) 段。

3.10.1 Near Code Segment

small 内存模式通常是汇编语言程序（不需要连结其它语言模块）的最佳选择。这种内存模式预设对于 code 和 data 是 near（双字节）地址。这可使程序执行较快，而且使用较小内存。

当你使用 .MODEL 和简化段伪指令，此 .CODE 伪指令会指定 assembler 开始一个 CODE 段。而下一个段伪指令会关闭前一个段。也就是说上一个段到此结束，接下来是下一个段的开头。END 伪指令是说明程序到此结束，指定 assembler 所有段到此结束，即所有的程序到此已结束，下面已无程序。

3.10.2 Far Code Segment

当你需要超过 64KB 的 code 段时，可使用 medium、large 或 huge 内存模式去建立 far 段 (大于 64KB 段)。

medium、large 和 huge 内存模式使用 far code 地址。在这些较大的内存模式中，此 assembler 会为每一个模块建立一个不同的 code 段。如果你在 small、compact 或 tiny 模式下，使用多个 code 段，此 Linker 会汇编所有模块的 .CODE 段而形成一个程序段。

对于 far 代码段，此 assembler 会为每一个代码段取一个 MODNAME _ TEXT 名称，这个 MODNAME 是这个模块的名称。而 near code ，assembler 会为每一代码段取节名为 _ TEXT，而使得 Linker 将所有模块的 code segment 集中成为一个代码段。你可以在 .CODE 之后加上一个新名称，去取代此名称。

对于 far code，一个单一模块可以包含多个代码段，可在每个 CODE 之后加上一个名称。例如下面例子建立两个不同的代码段，FIRST _ TEXT 和 SECOND _ TEXT。

```
.CODE FIRST
.
.                    ; First set of instruct here
.CODE SECOND
.
.                    ; Second set of instruct here
```

无论何时，处理器执行一个远程调用（far call）或转移（jump），都会将新的段地址载入 CS 中。

注意一般时候，assembler 总是假设 CS 寄存器包含目前代码段或 group 的地址。

3.11 使用 .STARTUP 和 .EXIT 去开始和结束 CODE

开始和结束一个 MS-DOS 的程序，最容易的方法是在主程序使用 .STARTUP 和 .EXIT 伪指令。在主程序中通常包含程序执行的起始点和终止点。你不需要在其它的模块中使用这些伪指令。

这些伪指令使得 MS-DOS 程序容易进行维护。他们自动产生适合 .MODEL 指定的堆栈距离的程序码。然而他们不能运用 32 位操作系统的 flat 模式的程序。因此你不应该在 WINDOWS NT 的程序中使用 .STARTUP 或 .EXIT。为了要启动一个程序，你可以在程序要开始执行的地方放置 .STARTUP，一般都是接在 .CODE 之后。

```
.CODE
        .STARTUP
        .
        .
        .EXIT
END
```

注意：.EXIT 会产生执行码，而 END 不会。此伪指令指定 assembler 已到程序的结尾。事实上所有的主程序（主模块）或子程序（其它模块），不论是使用简化或完整段都必须使用 END 伪指令结束。

如果你不使用 .STARTUP，你必须给 END 指定一个起始地址，实例如下：

实例 1：

```
.CODE
START:
.
; Place executable code here
.
END START
```

实例 2：

```
.CODE
main PROC
.
.
main endp
END main
```

只有 END 伪指令之后才有可能加程序的起始地址。当程序中已有 .STARTUP 出现，则 assembler 会忽略后面的参数，也就是在 END 之后可以不加程序的起始地址（如 START 或 main）。

对缺省的 NEARSTACK 属性，.STARTUP 会将 DS 指到 DGROUP 和设置 SS：SP 与 DGROUP 有关联，会产生如下代码：

```
@startup:
        mov dx, DGROUP
        mov ds, dx
        mov bx, ss
        sub bx, dx
        shl bx, 1; r higher, 可写成 shl bx, 4
        shl bx, 1
        shl bx, 1
        shl bx, 1
        cli         ; not necessary in .286 or higher
        mov ss, dx
        add sp, bx
        sti         ; not necessary in .286 or higher
        ...
        END @startup
```

一个 MS-DOS 程序有 FARSTACK 属性，就不需要去调整 SS：SP，所以 .STARTUP 只初始化 DS，如下：

```
@startup:
        mov DX, DGROUP
        mov DS, DX
        ...
        END @startup
```

当程序终止执行时，你可以向操作系统传回一个退出码 (exit code)。应用程序会检查这个 exit code，通常我们假设，exit code=0 表示无问题产生；exit code=1 表示有错误而终止了程序执行。.EXIT 伪指令可接受 1 个字节的退出码参数。

```
.EXIT 1         ; Return exit code 1
```

.EXIT 产生下列程序码并向 DOS 交还控制权并终止程序执行。而返回值可以是

一个常数、内存变量或1个字节的寄存器值（指定在AL中）：

```
MOV  AL，value
MOV  AH，04Ch
INT  21H
```

函数4Ch可用来结束一个程序，并传回我们在AL中指定的返回码。该函数执行后，DOS会作一些我们程序所忽略掉的“清理”操作；重置中断向量22H、23H和24H为原来的缺省值，清除文件缓冲区，关闭所有被打开的文件，并释放所有分配给程序的内存。

由于4Ch函数（结束程序执行）比中断20H或21H的00H函数更灵活且更好用，因此，6.11版以前的程序我们会很自然地选用函数4Ch来结束我们的程序，而现在6.11版只需使用.EXIT伪指令即可。

如果你的程序没有在AL中指定返回值，.EXIT将返回当时在AL中的任意值。

3.12 MOV指令

MOV指令可以说是汇编语言中最常用的指令，所以一定要熟记它的用法，下面我们会详细介绍各种用法，请各位仔细阅读，否则后面的章节就可能有麻烦了。

MOV　目的操作数，源操作数

源操作数与目的操作数可以是8、16或32位，并且要长度相同（size）。如果源操作数与目的操作数长度不同时，assembler会有操作数必须是相同大小的信息出现。MOV会将源操作数的内容送给目的操作数，所以源操作数的内容并不会改变，执行完后目的操作数的内容等于源操作数的内容。

可能使用的实例摘要说明：

```
(1) mov ax , bx          ; 寄存器←寄存器
(2) mov ax , num         ; 寄存器←变量
(3) mov num, ax          ; 变量←寄存器
(4) mov cx, 1234H        ; 寄存器←立即数
(5) mov num, 5678H       ; 变量←立即数
(6) mov al, 0FFh         ; 寄存器←立即数
```

若立即数的最高4位>9，则前须在最左边加"0"。

3.12.1 操作数的形式限制

1　CS与IP不能当作目的操作数。

2　立即数与段寄存器不能直接给段寄存器（CS、DS、SS、ES）传送数据。

源操作数与目的操作数必须是相同长度。若源操作数是一个立即数，目的操作数为8位时，立即数就不能超过0FFh（225）。若目的操作数为16位，立即数便不能超过0FFFFh（65535）。

3　变量也不能直接给变量传送数据，不过可以通过一个同样大小的寄存器做中间的媒介。

使用实例如下：

例 1：

不正确	正确	正确
mov var2, var1	mov AX, var1	mov AL, var3
	mov var2, AX	mov var4, AL

；※ 假设 var1、var2 为 16 位，var3、var4 为 8 位

例 2：

不正确	正确
move DS, @DATA	mov AX, @DATA
mov CS, 1	mov DS, AX

；※ 注意@DATA 代表数据段的地址，是一常量。

例 3：

不正确	正确
mov ES, DS	mov AX, DS
	mov ES, AX

在需要执行速度较快的场合，程序中的指令应多使用寄存器，因为寄存器的运算最快。所有的寄存器都可做为源操作数。除了 CS 与 IP 外，其余寄存器都可做为目的操作数。

一个立即数（即整数常数）可以给除了段寄存器和 IP（指令寄存器）外的所有寄存器传送数据。只要立即数不超过目的操作数的长度即可。

```
mov BL, 1    ;  正确
mov BX, 1    ;  也正确
```

3.12.2 直接寻址

在 MOV 指令中，变量的名字可做为操作数之一。源操作数若为变量，则会将变量所在的地址的内容送至目的操作数中。若目的操作数为变量，则会将源操作数的内容送至变量所在的地址中；称为直接寻址（CPU 会直接将变量地址的内容直接移入目的操作数中）。

下面的实例假设 num 变量为 1 个字节大小。

mov b1, num；如果 num=1，则 b1=num=1

mov num, b1；如果 b1=1，则 num=b1=1

当使用直接寻址时，也可以在变量名称后面加一个偏移量。比如将 array 改为 array+1，所存取的操作数则是 array 变量所在地址的下一个地址的内容。我们称 array 或 array+1 为一个有效地址（effective address；或 EA），编译程序会计算这个操作数的地址。该地址被用来存取在指定地址的内存。因为是直接寻址，所以在汇编时即可计算其地址。下面的实例假设 array 是一行数字。

```
array    BYTE    01H,    02H,    03H,    04H,    05H
```

01H	02H	03H	04H	05H
array+0	array+1	array+2	array+3	array+4

```
mov AH, array          ; AH=01H
mov AL, array+1        ; AL=02H
mov BH, array+2        ; BH=03H
```

```
mov BL, array+3          ; BL=04H
mov CL, array+4          ; CL=05H
```

3.12.3 PTR 运算符

有时我们可以使用 PTR 运算符帮助我们解释操作数的属性。语法：

type PTR

PTR 指明操作数真正的大小。下表列出所有 PTR 使用情况。

运　算　符	说　明
Byte ptr	Byte 操作数
Word ptr	Word 操作数
Dword ptr	Doubleword（32 位）操作数
Fword ptr	Farword（48 位）操作数
Qword ptr	Quadword（64 位）操作数
Tbyte	Ten-byte（80 位）操作数
Near ptr	Within the current segment
Far ptr	Outside the current segment

BYTE PTR 表示指定要存取单字节操作数，WORD PTR 表示指定要存取双字节操作数，使用实例如下：

```
mov word ptr word _number, 10
mov byte ptr byte _number,  5
mov ax, word ptr var1       ; Var1 is a word 操作数
mov bl, byte ptr var2       ; Var2 is a byte 操作数
call far ptr display        ; Call a far subroutine
; mov word _number, 10      ; illegal
; mov byte _number, 5       ; illegal
```

有时我们需要将 8 位的源操作数移入 16 位目的操作数的低 8 位或高 8 位时，或将 16 位源操作数移入 32 位的目的操作数的低 16 位或高 16 位时，便可借助 PTR 运算符帮助。

```
num1      word      1234H
num2      dword     12345678H
(1) mov byte ptr    num1,     0     ; num1=1200H
(2) mov byte ptr  num1+1,     0     ; num1=0034H
(3) mov word ptr    num2,     0     ; num2=12340000H
(4) mov word ptr  num2+2,     0     ; num2=00005678H
```

假如要存取某地址的内容，也可以将地址放在中括号内达到存取的目的。

```
mov AL, [100]      ; 将地址为 100 的内存的内容移至 AL
mov [100], AH      ; 将 AH 的内含移入地址为 100 的内存单元中
```

```
mov AX，[100]      ；将地址 [100]，[101] 共两个字节的数据移入 AX
```

由于寄存器操作数已暗示了操作数的大小，所以不需使用 PTR。

3.12.4 Offset 运算符

OFFSET 运算符可以从它所属的段开始算起的距离（偏移量；offset）返回（return）一个标记（label）或变量。目的操作数必定是 16 位寄存器。

```
array dB 01H，02H，03H，04H，…

mov SI，offset array ；SIpoints to array
mov AL，[SI]        ；AL=01H
```

上例中 Offset 运算符将 array 变量的起始地址移入 SI 寄存器中。我们说 SI 指到 array，因为它包含 array 变量的地址。当一个寄存器或变量储存一个地址，我们称它为指针（pointer）。利用它可以很容易地指到后面的地址。上例中使用 [SI] 方式，每次将 SI 加 1，即可轻松地存取一段连续内存中的数据。

3.12.5 SEG 运算符

SEG 运算符可返回一个标记（label）或变量的段值。我们通常用它将一个地址送至段寄存器。

```
mov AX，seg array；set DS to segment of array
mov DS，AX
```

使用 SEG 运算符可强迫将变量 array 的段地址移入 AX，通过 AX 便可将 array 所属的段地址移入 DS 中。

3.12.6 XCHG 指令

XCHG 指令交换两个寄存器的内容或一个寄存器和一个变量。语法：

```
XCHG op1，op2          ；op1 与 op2 可为寄存器或变量
```

XCHG 指令可将 op1 与 op2 两操作数内容交换。只是其中必须有一个操作数为寄存器。源操作数与目的操作数必须相同。下列都是合法的指令：

```
xchg      ax，      bx
xchg      ah，      bl
xchg     var1，     bx
```

在排序的过程中常须做数据交换操作，如果使用 XCHG 指令，即可减少使用寄存器或变量作暂时储存数据的次数，提高速度。

例：num1 与 num2 交换

```
num1  BYTE  11H
num2  BYTE  22H
```

（1）不用 XCHG 指令	（2）使用 XCHG 指令	（3）不合法
mov AL，num1	mov AL，num1	XCHG num1，num2
mov AH，num2	XCHG AL，num2	
mov num2，AL	mov num1，AL	
mov num1，AH		

※ 执行后 num1=22H，num2=11H。

3.13 PUSH 与 POP 指令

PUSH　源操作数

POP　目的操作数

操作数必须是 16 位的寄存器或变量。

PUSH 指令执行时，CPU 会将 SP 自动减 2，将源操作数压入 SP 所指地址的堆栈中。使用 PUSH 指令的好处就是在某些场合需要保存寄存器旧值，可利用 PUSH 指令将其暂时储存在堆栈中，最后再用 POP 指令，将旧值弹出到目的操作数中，SP 再自动加 2，以便指到堆栈顶端尚未弹出的数据地址。

SP 会加减 2 是因为源操作数或目的的操作数必为 16 位的寄存器或变量，千万不要压入单字节的寄存器或变量，或弹出到单字节的寄存器或变量。

例：

```
source word 1234H
dest   word 0ABCDH
(1) 合法          (2) 不合法
push   AX         push   AL
push   source     pop    BL
pop    AX         pop    CS
pop    dest       pop    IP
```

POP 指令的目的操作数不能为 CS 或 IP。

第4章　地址与指针

本章描述如何初始化缺省的段寄存器去存取 near 与 far 地址、code 与 data 以及相关的 operators、syntax、displacements。在最后的章节中会解释如何使用 TYPEDEF 伪指令去说明指针与 ASSUME 伪指令，去告诉 assembler 有关寄存器包含指针的信息，有些以※为开头的主题是属于高级的问题，您可选择参考阅读。

MASM 程序在实模式（real mode）下执行，存取程序和数据时需要段地址。段地址中的程序和数据的地址是和在一段寄存器中的段地址有关联的。你可以使用指针去存取汇编语言程序中的数据。（指针是指一个变量；包含它的值的一个地址）。

4.1　段的地址

在你的程序中要使用段的地址之前，你必须先初始化段寄存器。初始化过程根据你要使用的寄存器与你所选择的简化段伪指令或完全段定义。简化段伪指令会为你处理大部分的初始化程序。本章将解释如何去指定编译程序及处理器（CPU）段的地址，与如何存取在这些段中的近程（near）及远程（far）的程序的数据。

4.1.1　初始化缺省的段寄存器

8086 家族处理器的段结构并不需要你每次存取内存时，都指定两个地址（segment 与 offset）。8086 家族处理器使用一个系统缺省的段寄存器去简化存取大部分公用的数据与程序。

在程序的开头，段寄存器 DS、SS 和 CS，正常情况下是初始化至缺省段。如果你的主程序是用高级语言编写，那么会自动初始化这些段寄存器。如果你的主程序是用汇编语言写成，你必须自己初始化这些段寄存器。按照下列的步骤去初始化段：

1　编译程序哪一段和寄存器有关联，编译程序在编译时必须知道这些缺省段。

2　写需要的程序去处理段寄存器，处理器哪一个段是和寄存器相关联。

这些步骤将在下面的章节中介绍。

4.1.2　指定和编译程序相关的段值

初始化段的第一步是告诉编译程序哪一段和寄存器有关联。你可以利用 ASSUME 伪指令。如果你使用简化段伪指令，编译程序会自动产生适当的 ASSUME 指令。如果你使用完整段说明，除了 CS 之外，你必须自己编写 ASSUME 指令，去指定相关的寄存器。

.STARTUP 伪指令产生 startup code 并且设 DS 与 SS 相等（除非你指定 FARSTACK），允许通过 SS 或 DS 去存取缺省的数据。这可加快由 assembler 产生的程序效率。* DS＝SS 的习惯可能不能在某种应用程序下工作，如在 MS-DOS 中的常驻程序和 Windows 动态链结库。你可以在程序中使用完整段定义及使用相同的程序码去设置 DS＝SS。

这里有使用完整段定义 ASSUME 的例子。

ASSUME cs：_TEXT，ds：DGROUP，ss：DGROUP

这个例子和由使用简化段伪指令（.STARTUP）产生的 ASSUME 指令相等（small 模式；

NEARSTACK)。当然对于数据和程序也可以有不同的段。也可以使用 ASSUME 去设置 ES，实例如下。

```
ASSUME    CS：MYCODE，ds：MYDATA，ss：MYSTACK，es：OTHER
```

正确使用 ASSUME 可以帮助查找寻址错误。比如使用 .CODE，编译程序假设 CS 是当前的段。当你使用简化段伪指令 .DATA，.DATA?，.COnST，.FARDATA，或 .FARDATA?，编译程序会自动假设 CS 是 ERROR 段。这可防止指令不要出现在这些段中。

如果你用完整段定义，你可以在数据段中通过使用 ASSUME CS：ERROR 去完成相同的工作。

无论是使用简化或完整段，你都可以通过假设 NOTHING 去取消 ASSUME 的控制。例如你可以使用下面叙述去取消先前对 ES 的假设：

```
ASSUM    es：    NOTHING
```

若使用 .MODEL，编译程序会对（DS、ES、SS）设置 ASSUME 至当前段。

4.1.3 指定处理器相关的段值

在初始化段的第二步和最后一步是去指定处理器在执行时（run time）的段值。在执行时间内如何初始化段值，是按照操作系统和你所使用的简化段伪指令，或是完整段定义。

1. 指定起始地址

程序的起始地址是决定开始执行的地方。在操作系统载入一个程序之后，它会简单地跳至起始地址，并转移处理器的控制权给程序。真正的地址只有载入程序（Loader）知道；该程序只决定在一尚未决定的程序段地址的偏移量。这就是为什么正常的应用程序通常是参考到一个所谓的“relocatable code”，因为它执行时忽略这 loader 在内存中放置该程序的地方。

起始地址的偏移量还须视程序的形式而定。EXE 文件的程序包含一个 header，它会从 loader 读入偏移量和结合段去形成一个起始地址。COM 文件的程序没有这样的 header，它是通过 loader 跳至一个程序的第一个字节。

.STARTUP 伪指令（表示开始执行的地方）提供给你使用简化段伪指令，对于 EXE 程序，在你开始执行的第一条指令之前应立即放置 .STARTUP。对于 COM 程序，在你源文件中的第一个汇编指令之前放置 .STARTUP。

如果你是使用完整段伪指令或不喜欢使用 .STARTUP，你必须有两个步骤去指明起始地址：

（1）标记此起始指令。

（2）在 END 伪指令提供相同的标号。

这些步骤是告诉 Linker 程序开始执行的地方。下面的实例列出了这两个步骤（tiny model 程序）。

```
-TEXT SEGMENT WORD PUBLIC 'CODE'
    ORG     100h ; Use this declaration for . COM files only
start: .            ; First instruction here
       .
       .
_TEXT ENDS
       END start; Name of starting label
```

在一个 tiny model 的程序中（没有使用 .STARTUP 伪指令），ORG 会在程序段的偏移 100h（此实例为 100h）之处放置第一条指令，为了要建立一个 256 字节（100H）的数据段，称为代码段前置区 (program Segment Prefix；PSP)。操作系统会处理初始化 psp 的工作，所以你只要确定该段的存在。

2. 初始化 DS

如果你使用 .STARTUP 或你所编写的是一个 Windows 程序，该 DS 是自动初始化成正确值。如果在 MS-DOS 下没有使用 .STARTUP，你必须使用下列的指令去初始化 DS 以确保能正确存取数据段的数据。

```
MOV     ax,     DGROUP
MOV     ds,     ax
```

该初始化的工作需两条指令，因为该段名（DGROUP）是一个常数，编译程序不允许常数值直接载入至段寄存器。上面的例子是载入 DGROUP，但是你也可以载入任何有效的段或组。

3. 初始化 SS 和 SP

如果你使用 .STACK 简化段伪指令或定义一完整段时使用 STACK 连结形式，该 SS 和 SP 会自动开始设置它的初值。使用 .STACK 伪指令初始化 SS 至堆栈段。如果你要让 SS 等于 DS，使用 .STARTUP。对一个 EXE 文件而言，该堆栈是决定于载入时间。对于一个 COM 文件，该载入程序（loader）会设置 SS 等于 CS，以及 SP 会设为 0FFFEH。

如果你的程序并不存取远程数据（FAR　DATA），你不需要设置 ES 寄存器的初值。如果你选择去初始化，可以使用和对 DS 寄存器一样相同的技巧。你也可以使用相同方法去初始 SS 至 far stack。

4.2　近程与远程地址

和暗示的段名字或段寄存器有关联的地址被称为近程地址(near address)；而和一明确的段有关联的地址被称为远程地址（far address)。编译程序会自动处理近程和远程的程序，详细内容将在下面的章节描述。不过你必须指定如何去处理远程数据。

Microsoft 段模式会在一个称为 DGROUP 的组中放置所有的近程数据和堆栈。属于近程的程序是放置在一个称为 TEXT 的段中。每个模块的远程程序或远程数据是各自被放置在分离的段中。编译程序不能决定许多程序实际的地址。编译程序产生一预备的记录，而连结程序提供这个地址（一旦它已决定所有段的位置)。通常一个重定位的操作数会参考一个标记，但是也有例外。在下两节的实例包含许多有关重定位的近程与远程数据。

4.2.1　Near Code

在近程程序（near code）中要执行控制权转移不需要改变段寄存器。当控制流程指令，比如 JMP、CALL、RET 使用时，该处理器会自动处理改变在 IP 寄存器的偏移。这叙述改变 IP 寄存器到新的地址，但是留 call nearproc；Change code offest 下段没有改变。当子程序 return 时，处理器会重设 IP 到 CALL 指令之后的下一个指令的偏移。

4.2.2　Far Code

当要处理远程程序（far code）时，处理器会自动处理段寄存器的改变。语句 call farproc；

Change code segment and offset 会自动搬移 farproc 子程序的段与偏移到 CS 和 IP 寄存器。当从子程序 return 时，处理器会重设 CS 与 IP 到 CALL 指令之后的下一个指令的代码段和偏移。

4.2.3 Near Data

程序可以直接存取近程数据（near Data），因为段寄存器已经掌握了数据项目的正确段。“near Data”名词通常是被使用去存取在 DGROUP 群组内的数据。

第一次初始化 DS 和 SS 寄存器之后，这些寄存器正常是指到 DGROUP。如果你在程序的执行期间修改任一个寄存器的内容，在存取任何 DGROUP 内的数据之前你必须再重新载入 DGROUP 的地址至寄存器。处理器假设所有存储体存取都与 DS 寄存器内的段值有关联，除了使用 BP 或 SP 存取的例外。处理器会结合这些寄存器和 SS 寄存器。下面的例子说明了处理器如何选择 DS 或 SS，这依赖于这些指针操作数是否包含 BP 或 SP。注意当 DS 与 SS 是相等时，该区别将失去意义。

```
nearvar WORD 0
        .
        .
        .
        mov  ax, nearvar      ; Reads from DS: [nearver]
        mov  di, [bx]         ; Reads from DS: [bx]
        mov   [di], cx        ; Writes to   DS: [di]
        mov   [bp+6], ax      ; Writes to   DS: [bp+6]
        mov  bx [bp]          ; Reads from SS: [bp]
```

4.2.4 Far Data

要读或修改一远程地址，段寄存器必须指到数据段。这需要两个步骤。首先载入段（正常不是 ES 就是 DS）的正确值，然后（选择性地设置一假设的段寄存器到段的地址。

Flat 模式不需要远程地址，是缺省值，所有地址都是对应到段寄存器的初始值。因此远程地址这章节不适用于 flat 模式程序。

一般要存取远程地址的方法是初始化 ES 寄存器。下面的实例列出了两个方法：

```
; First method
   mov ax, SEG farvar        ; Load segment of the far address
   mov es , ax               ;    into ES
   mov ax, es: farvar        ; Provide an explicit segment
                             ;    override on the addressing
; Second method
   mov ax, SEG farvar2       ; Load the segment of the far
   mov es, ax                ;    address into ES
  ASSUME ES: SEG farvar2     ; Tell the assembler that ES
                             ;    points to the segment
                             ;    containing farvar2
   mov ax, farvar2           ; The assembler provides the ES
```

```
;       override since it knows that
;       the lable is addressable
```

在载入段的地址到 ES 段寄存器之后，你可以明确地重设段寄存器，所以这地址是正确的（方法一），或允许编译程序为你插入强制（override）指令（方法二）。编译程序使用 ASSUME 叙述去决定哪一个段寄存器去寻址一存储体段。为了要使用段强制运算符，左边的操作数必须是一段寄存器，而不是一段名称。

如果一条指令需要强定段的话，这最后的程序稍会大一点也较慢。因为重定段需要将指令重新编码。然而，这还是比要重复载入预设的段寄存器的程序码还较小。

DS、SS、FS、GS 段寄存器（FS 与 GS 只可用在 80386/486 的处理器）也可以被使用去寻址其它的段。

如果一个程序使用 ES 去存取远程的数据，执行完后不需要再恢复 ES 寄存器之值。然而许多 compiler 需要你在返回到用高级语言写成的模块（或子程序）之前恢复 ES。

为了存取远程数据，首先设置 DS 到此远程段，执行完后再恢复原先的 DS 之值。使用 ASSUME 伪指令去让编译程序知道 DS 不再指向这预设的数据段，表示如下：

```
push ds                         ; Save original segment
mov as, SEG fararray            ; Move segment ino data register
mov ds, ax                      ; Initialize segment register
ASSUME ds: SEG fararray         ; Tell assembler where data is
mov ax, fararray [0]            ; Set DX: AX=dword variable
mov dx, fararray [2]            ;       fararray
.
.
.
pop ds                          ; Restore segment
ASSUME ds: @ DATA               ;    and default assumption
```

"直接存储体操作数"，描述了另一个存取远程数据的方法。在先前的重设 DS 的技巧实例是对于要存取一连串远程数据最好的方法。当只要存取一个或两个远程的变量时，强定段的方法是较好的。

如果你的程序改变 DS 去存取远程数据，在执行完后应该要恢复 DS，这样可以允许子程序假设 DS 是近程数据的段。许多 compilers，包含 Microsoft compilers 习惯这样使用。

4.3 运算符（Operator）

汇编语言的语句中通常包括指令，操作数和运算符。下表列出经常在汇编语言中使用到的运算符：

运算符（Operator）	说　明
算术运算符（Arithmetic）	+，－，*，/，（），MOD。 ※（MOD 为取余数之意）

（续）

运算符（Operator）	说　明
布尔运算符（Boolean）	NOT，AND，OR，XOR
索引运算符（Index）	[N] 指定存取第 N 个地址的值
偏移运算符（Offset）	Offset 运算符可取得变量的偏移
段运算符（SEG）	传回一个变量或标记的段值
PTR 运算符	强迫将一特定值给目的操作数
SHORT 运算符	表示一标记在（－128～127）字节的范围内

算术运算符：包括＋，＝，＊，/，MOD，（）

实例	结果	实例	结果	实例	结果
7＋2	9	6＋2＊1	8	－5－5	－10
6＊2－1	11	4＊2	8	(7－5) ＊1	2
5/2	2	1＊3 Mod 4	3	82 mod 8	2
'A' －41H	0	－(53 mod 5) ＊7	－21		

由上例可知优先权（有较先执行的权利）顺序如下：

（），正负号，＊，/，mod，＋，－

优先权高　优先权低

※其余的运算符将在后面的章节中陆续介绍。

4.4 操作数（Operand）

汇编语言指令所要处理的数据，称为操作数（Operand）。有汇编程序的操作数出现在指令右侧的操作数位置。

这一节描述了四种指令操作数：register、immediate direct memory，indirect memory。许多指令，如 POPF 与 STI 中没有操作数出现在操作数位置。

某些其它指令如 NOP 和 WAIT，只会影响处理器控制，所以并不需要操作数。

下列四种操作数类型在其它的章节描述：

Operand Type	Addressing Mode
Register	8 位或 16 位寄存器（8086～80486）；也可以是 32 位寄存器（80386/486）
Immediate	包含在指令本身中的一个常数值
Direct memory	在存储器中一固定的位置
Indirect memory	一个在运行时才能决定的存储器位置。这地址可能储存在一个或两个寄存器中

指令若有两个操作数，一般总是由右到左操作。右边的操作数是源操作数，表示这个数据在运算的过程中只读而且不会改变。左边的操作数是目的操作数，这个数据将被影响而且可能被指令改变。

4.4.1 Register 操作数

寄存器操作数参考储存在寄存器内的数据。下面是典型的寄存器操作数实例：

```
mov bx, 10          ; Load constant to BX
and ax, bx          ; ADD BX to AX
jmp di              ; Jump to the address in DI
```

储存在基址（base）或索引（index）寄存器内的偏移通常做为指向内存的指针之用。你可以储存偏移在基址或变址寄存器之内，然后使用这个寄存器当做一个间接内存操作数(indirect memory 操作数)。例如：

```
mov [bx], dl    ; Store DL in indirect memory 操作数
inc  bx         ; Increment register 操作数
mov [bx], dl    ; Store DL in new indirect memory 操作数
```

前面的实例移动在 DL 内的值到由 BX 所指的 2 字节的连续内存地址。任何会改变寄存器值的指令也会改变由寄存器所指的数据项目。

4.4.2 Immediate 操作数

立即操作数 (Immediate 操作数) 是一个常数或一个常数运算式的结果。编译程序会在编译时（assembly time）重新将立即数编码在指令中。这里有三个典型的立即数实例：

```
mov cx, 20          ; Load constant to register
add var, 0fH        ; Add hex constant to variable
sub bx, 25 * 80     ; Subtract constant expression
```

目的操作数不允许立即数出现。如果源操作数是一个立即数，那么目的操作数不是一个寄存器就是一个直接内存变量（或称内存地址），因为要提供一个储存运算结果的位置。

立即运算式通常包含使用 OFFSET 和 SEG 运算符，描述在下列段落中。

4.4.3 OFFSET 运算符

地址常数是一个立即操作数的特别型式，它包含一个偏移量或段值。OFFSET 运算符会回传（return）一个内存地址（变量）的偏移，如下：

```
mov bx, OFFSET var ; Load offset address
```

因为在不同模块中的数据可能属于单一段，对于每一模块来说，编译程序并不知道在一个段中的真正偏移。因此，var 虽然是一个立即操作数，但偏移还是直到 Link 时才知道。

4.4.4 SEG 运算符

SEG 运算符会回传（return）内存地址的段值：

```
mov ax, SEG farvar ; Load segment address
mov es, ax
```

一个特别的段真正的值要直到程序被载入内存才知道。对于 .EXE 文件，Linker 会在程序的 header 中列出所有 SEG 运算符出现的地址列表。Loader 会读这个列表和填满在每一地址所需要的段地址。因为 .COM 文件没有 header，编译程序不允许在一个 tiny 模式程序有再定段运算式。

SEG 运算符也可以传回一个变量的“frame”（如果它出现在指令）。“frame”是一个段（segment）、群组（group）或非外部变量的强定段（segment override）的值。例如指令

```
mov ax, SEG DGROUP: var
```

该指令放置 var 被配置的 DGROUP 值在 AX 中。如果你没有包含“frame”，SEG 会传回这个变量的群组值（如果群组存在）。如果变量没有定义在一个群组中，SEG 将传回该变量的段地址。

4.4.5 Direct Memory 操作数

直接操作数载明在一个指定地址中的数据。指令是作用于地址的，不是地址本身。除了当大小是被另一操作数暗示外，你必须指明直接操作数的大小，这样指令才能存取到正确的内存数量。这下面的实例显示如何用 BYTE 假设明确地指明数据的大小：

```
.DATA?                  ; Segment for uninitialized data
    var BYTE?           ; Reserve one byte, labeled "var"
.CODE
    ...
    mov var, al         ; Copy AL to byte at var
```

任何内存地址都可以做为一个直接操作数，只要大小是被指明（或暗示）而且位置是固定的。在地址内的数据是可以改变的，但是地址不能改变。缺省值是使用直接寻址的指令是使用 DS 寄存器的。你可以通过使用任何下列的运算符之一去建立一个指向内存地址的运算式。

Operator Name（运算符名称）	Symbol（符号）
Plus	+
Minus	—
Index	[ ]
Structure member	.
Segment override	:

这些运算符在下面的章节详细讨论。

1. Plus, Minus, and Index

.Plus（“+”）与 Index（“[ ]”；索引）运算符应用在直接操作数时，执行的结果事实上是一样的。例如，下列两条语句从一个数组搬移第二个 word 值到 AX 寄存器：

```
mov ax, array [2]
mov ax, array+2
```

这个索引运算符可以包含任何直接操作数。下列语句是相等的：

```
mov ax, var
mov ax, [var]
```

许多程序员习惯将操作数用括弧括起来。

minus（“—”）运算符的作用和你所想的一样。下面两条语句（执行情况一样）可以查询 array 之前两个字节位置的值。

```
mov ax, array [-2]
mov ax, array-2
```

2. 结构段（Structure Field）

运用结构运算符（.）可以参考一结构或区段（“Field”）的一特别元素。就如同使用 C 一样。

```
mov bx, structvar.field1
```

这个结构操作数的地址是 structvar 与 field1 的偏移总和。

3. （Segment Override）运算符

段强定运算符（:）指明一段部份地址，而不是使用预设的段。当在指令使用时，这个运算符可应用为段寄存器或段名称：

```
mov ax, es: farvar ; Use segment override
```

如果预设段是很明显已提供的话，编译程序将不会产生段强定。因此，下列两个语句事实上是以同样方式编译的：

```
mov [bx], ax
mov ds: [bx], ax
```

一段名称强定或使用段强定运算符指明这个操作数为一地址表示式。

```
mov WORD PTR FARSEG: 0, ax      ; Segment name override
mov WORD PTR es: 100h, ax       ; Legal and equivalent
mov WORD PTR es: [100h], ax     ; expressions
; mov WORD PTR [100h], ax       Illegal, not an address
```

以上实例显示，一常数表示式不可以为一个地址表示式，除非它有一段强定。

4.4.6 Indirect Memory 操作数

像直接操作数一样，间接操作数载明一指定地址的内容。然而，处理器在运行时通过参考寄存器的内容计算这个地址。因为在寄存器内的内容在运行时可以改变，所以间接操作数可以动态（dynamic）存取内存。

间接操作数使运行时的运算当成如间接的指针成为可能，以及动态索引数组的元素，包含索引多维数组。这章节将介绍在各个模式中间接操作数的各种特征。

4.4.6.1 16 位和 32 位寄存器间接操作数

对于间接操作数有许多规则与选项可以应用，通常可以忽略这个寄存器的大小。例如，对于间接操作数你必须总是指定寄存器与操作数的大小。但是你也可以使用多种方法去表示一间接操作数。这个章节描述了应用 16 位和 32 位寄存器模式的规则。

1. 指定间接操作数

索引运算符指明了对于间接操作数的寄存器。处理器使用被寄存器所指的数据。例如，下列的指令搬移在 DS：BX 地址的 word 值到 AX。

```
mov ax, WORD PTR [bx]
```

当你指定过一个寄存器，处理器在决定有效地址（为了要运算这数据的地址）时会相加两个地址的内容。

```
mov ax, [bx+si]
```

2. 指定替代值（displacements）

你可以指定一个地址的替代值，它是一个会增加有效地址的常数值。直接内存指定是最常见的替代值：

```
mov ax, table [si]
```

在这个重定位的表示式中，替代值 table 是一数组的基址地址；SI 是一数组元素的索引值。SI 的值在运行时被计算，通常在一个循环中。而载入 AX 的元素须视这指令执行时 SI 内的值。

每一个 displacement 可以是一个地址或数值常数。如果有超过一个 displacement，编译程序在汇编时会将之全部加起来，并重新编码，例如，在这个语句

```
tabel WORD 100 DUP (0)
      .
      .
      .
mov ax, table [bx] [di] +6
```

table 和 6 是 displacement。编译程序会把 6 这个值加到 table 中去得到全部的替代值。然而，这条语句

```
; mov ax, meml [si] +mem2
```

是不合法的，因为它尝试使用单一命令去结合两个不同地址的内容。

3. 指定操作数大小

指定操作数大小你可以下面三种方法来指明间接操作数的大小。

①借助变量说明时的大小

②使用 PTR 运算符

③已由其它操作数暗示大小

下面的实例列出了三种方法。假设 table 数组说明时大小为 WORD。

```
mov table [bx], 0              ; 2 bytes-from size of table
mov BYTE PTR table, 0          1 byte-specified by BYTE
mov ax, [bx]                   ; 2 bytes-implied by AX
```

4. 语法选择

编译程序对间接操作数的使用提供了多种的选择。然而，所有的寄存器都必须在中括号内。你可以将每一寄存器都用中括号括起来，或将所有的寄存器放在一中括号内，再用“+”运算符隔开。下列五个例子都是合法的，且其所代表的意义皆一样：

```
mov ax, table [bx] [di]
mov ax, table [di] [bx]
mov ax, table [bx+di]
mov ax, [table+ bx+di]
mov ax, [bx] [di] +tabel
```

以上所有的语句都是将在 table 索引值为 BX+DI 的值搬移到 AX。

5. 累加索引值

要存取到正确的数组数据元素必须精确调整变址寄存器的值，指向数组的索引寄存器值通常是调整为以零为基址的数组（如 C 语言的数组索引值也是由零开始），且按照数组元素的

大小来累加其值。对于一个 WORD 数组（即数组中的元素大小为一 WORD；2 bytes），中括号内的项目数必须乘二（可以通过左移一位达到乘二的目的）才能正确地指到所要的数据。

```
mov bx, 5                    ; Get sixth element（由零算起）
sh1 bx, 1                    ; 累加 2（word size），bx=10
inc wordtable [bx]           ; Increment sixth element in table
```

当你使用 80386/486 处理器的 32 位寄存器，你可以将累加值放在操作数中达到快速计算索引值和方便索引数组的数据的目的。详细的内容在“32 位寄存器间接操作数”章节中介绍。

6. 存取结构元素

结构运算符可以被使用在间接操作数中去存取结构元素。在这个例子中，结构运算符载入 students 数组第四个元素的 year 区段至 AL：

```
STUDENT    STRUCT
    grade      WORD     ?
    name       BYTE     20     DUP      (?)
    year       BYTE     ?
STUDENT    ENDS

students              STUDENT ( )
        .
        .                                          ; Assume array is initialized
    mov  bx,   OFFSET  students                    ; Point to array of students
    mov  ax,   4                                   ; Get fourth element
    mov  di,   SIZE  STUDENT                       ; Get size of STUDENT
    mul  di                                        ; Multiply size times
                                                   ;    elements to polint DI
                                                   ;    to current element
    mov  di,   ax
    mov  al,    (STUDENT PTR [bx+di] .year
```

4.4.6.2 16 位寄存器间接操作数

对于以 8086 为基础的计算机和 MS-DOS，你必须遵循由 8086 处理器建立的严格索引规则。只有四个寄存器（BP、BX、SI、DI）允许结合在一起使用。

BP 和 BX 是基址寄存器，SI 和 DI 是变址寄存器。你可以单独使用一个基底或变址寄存器，但如果你要结合两个寄存器，一定一个是基址，另一个是索引才可。下列前四个实例是合法的，后两个是不合法的：

```
  mov ax, [bx+di]        ; Legal
  mov ax, [bx+si]        ; Legal
  mov ax, [bp+di]        ; Legal
  mov ax, [bp+si]        ; Legal
; mov ax, [bx+bp]        ; Illegal-two base registers
; mov ax, [di+si]        ; Illegal-two index registers
```

表 4-1 列出你可以指定的间接操作数的寄存器。

表 4-1

Mode	Syntax	Effective Address
Register indirect	[BX]	Contents of
	[BP]	register
	[DI]	
	[SI]	
Base or index	displacement [BX]	Contents of
	displacement [BP]	register plus
	displacement [DI]	displacement
	displacement [SI]	
Base plus index	[BX] [DI]	Contents of base
	[BP] [DI]	register plus
	[BX] [SI]	contents of index
	[BP] [SI]	register
Base plus index	displacement [BX] [DI]	Sum of base
with displacement	displacement [BP] [DI]	register, index
	displacement [BX] [SI]	register, and
	displacement [BP] [SI]	displacement

不同的结合方式会有不同的 timings。

4.4.6.3 32 位寄存器间接操作数

在 80386/486 处理器上所写的指令可以使用 16 位或 32 位的段，但在这两种情况下使用的间接操作数是不同的。

在 16 位实际模式中，80386/486 的操作和在以 8086 为基础的处理器中一样，但有一点不同：可以使用 32 位寄存器。如果 80386/486 处理器是 enable（使能：使用 .386 或 .486 假指令），32 位一般寄存器可以使用在 16 位或 32 位的段。32 位寄存器减少了许多 16 位间接操作数的限制。你可以使用 80386/486 的许多特性去使得你的 MS-DOS 程序执行得较快且有效率（如果你准备牺牲与较早期的处理器的兼容性）。

在 32 位模式，一个偏位移可定址到 4GB（段还是以 16 位表示）。这有效地缩减了每一段的大小限制，因为很少有程序需要 4GB 的内存。

1. 80386/486 Enhancements

80386/486 处理器允许任何一般 32 位寄存器当做基址或索引寄存器，除了 ESP（只可做为基址不可做为变址寄存器）。然而还是不可结合 16 位和 32 位寄存器在一起。实例如下：

```
add edx, [eax]          ; Add double
mov dl, [esp+10]        ; Copy byte from stack
```

```
dec WORD PTR [edx] [eax]    ; Decrement word
cmp ax, array [ebx] [ecx]   ; Compare word from array
jmp FWORD PTR table [ecx]   ; Jump into pointer table
```

2. 累加因子（Scaling Factors）

在 80386/486 的寄存器里，变址寄存器可以有一累加因子（1、2、4 或 8）。任何寄存器除了 ESP 之外，都可做为变址寄存器和累加因子。

为了指定累加因子，使用（*）运算符。

你可以使用累加因子去索引不同大小元素的数组。例如，累加因子 1 适合于 byte 数组（没有累加因子的需要），2 适合于 word 数组，4 适合于 doubleword 数组，8 适合于 quadword 数组，这使我们能较方便的计算索引值。当然须牺牲一点执行时间。实例如下：

```
mov eax, darray [edx * 4]          ; Load double of double array
mov eax, [esi * 8] [edi]           ; Load double of quad array
mov ax, wtb [ecx+2] [edx * 2]      ; Load word of word array
```

因为在较早期的处理器没有此累加因子功能提供，所以必须再利用其它的指令去计算正确的索引值，以达到间接操作数的目的。你可将这章节中内容和“16 位寄存器间接操作数”的“累加索引值”章节相互比较。基本上，若基址寄存器是 EBP 或 ESP，预设的段寄存器是 SS。然而如果 EBP 被累加，处理器将视它为一变址寄存器且是相对于 DS 不是 SS。

所有的基址寄存器是相对于 DS。如果有两个寄存器一起使用，只能一个有累加因子。而这个有累加因子的寄存器是被定义为变址寄存器。而另一个定义成基址寄存器。如果没有使用累加因子，则第一个寄存器是基址寄存器。倘若只有一个寄存器被使用且没有使用累加因子，则视为基址寄存器且必须考虑它的预设段（若为 EBP 或 ESP 则为 SS）。倘若只有一个寄存器被使用且有使用累加因子，则按照前面的定义当然视为变址寄存器。

下面的例子将告诉你如何决定基址寄存器：

```
mov eax, [edx] [ebp * 4]     ; EDX base (not scaled-seg DS)
mov eax, [edx * 1] [ebp]     ; EBP base (not scaled-seg SS)
mov eax, [edx] [ebp]         ; EDX base (first-seg DS)
mov eax, [ebp] [edx]         ; EBP base (first-seg SS)
mov eax, [ebp]               ; EBP base (only-seg SS)
mov eax, [ebp * 2]           ; EBP * 2 index (seg DS)
```

3. 16 位与 32 位寄存器混用

汇编语言语句可以混合使用 16 位和 32 位寄存器。例如，下面的叙述是合法的；

```
mov eax, [bx]
```

这叙述搬移由 BX 所指的 32 位的值以 EAX。虽然 BX 是一个 16 位的指针，但它还是可以指到一个 32 位的段。然而，下面的语句是不合法的，因为不可以使用 CX 寄存器当成一个 16 位指针。

```
; mov eax, [cx] ; illegal
```

混合 16 位和 32 位寄存器当成操作数也是不合法的：

```
; mov bx, [eax]
```

这条语句搬移由 EAX 所指的 16 位值到 BX 寄存器。这可工作在 32 位模式。然而在 16 位

模式，搬移一个 32 位指针到 16 位段是不合法的。如果 EAX 包含一个 16 位值（即 32 位的高两个 bytes 是零），这条语句将可正常工作。然而如果 EAX 寄存器的高两个 bytes 不为零，这个操作数所指的段内容将不存在，会产生错误。如果使用 32 位寄存器当成 16 位模式的索引，你必须确定这个索引寄存器包含一个有效的 16 位地址。

4.5 程序堆栈

堆栈是一暂时储存数据的内存段。不像其它的段储存数据从低内存地址开始储存数据，堆栈储存数据从高内存地址开始储存数据。数据总是从堆栈的顶端通过“PUSH”或“POP”的操作进行数据处理。你必须通过堆栈的顶端来增加或移出数据。从堆栈移出数据称为“POP”最开头的两位。反之亦然。堆栈通常是参考一个 LIFO buffers，通过它做 last-in-first-out（后进先出）运算。

堆栈是任何重要程序的一个非常重要的部份。程序时常会使用堆栈去暂时储存返回地址、子程序参数、内存数据或寄存器。

SP 寄存器可以作为一存取堆栈顶端的间接操作数。首先，堆栈是一未具初始化的段（有限制大小）。当你的程序增加数据到堆栈，堆栈是由高内存地址往低内存地址成长的。当要从堆栈移出数据时，刚好是由相反方向缩减的。

4.5.1 储存在堆栈的操作数

PUSH 指令存储操作数在堆栈。POP 指令取回最近被 pushed 的值。当一值被 pushed 入堆栈，编译程序会将 SP（Stack Pointer）寄存器处减 2。在 8086-based 处理器中，SP 总是指到堆栈顶端。PUSH 与 POP 指令使用 SP 寄存器去追踪当前的位置。当一值被 poped 出堆栈，编译程序会将 SP（Stack Pointer）寄存器处增 2。因为堆栈总是包含 word 值，所以 SP 总是以 2 的倍数改变。当 PUSH 或 POP 指令执行在 32 位的代码段（也就是有使用 USE32），编译程序移动 4-byte 值及 ESP 以 4 的倍数改变。

8086 与 8088 处理器和较晚期的 Intel 处理器在他们如何 push 和 pop SP 寄存器是不同的。如果你在 8086 或 8088 上执行 push sp 指令，则压入堆栈的值是 push 运算之后的 SP 值。也就是执行前 SP 值是 2，执行完后 SP＝0，而被压入堆栈的是零不是我们所想的 2。

在 8086，PUSH 与 POP 只能使用寄存器或内存运算式当做操作数。其它的处理器允许一个立即数去做 PUSH 的操作数。例如，下列的语句在 80186～80486 处理器中是合法的：

```
push    7       ;    3 clocks    on    80286
```

上例的语句比下列相等的语句较快，而在 8088 或 8086 却需下列的叙述：

```
mov ax, 7 ; 2 clocks plus
push ax   ; 3 clocks on 80286
```

从堆栈 pop 出的 word 数据是以反转的顺序操作：最后 push 的数据先被 pop 出来。如果你想要回到堆栈执行前的状态，你执行 push 的次数一定与 pop 的次数相同。如果你恢复堆栈的状态而不再使用其中的值，你可将 SP 寄存器的值减掉正确的 word 数目。

若要存取在堆栈中的操作数，记住通过使用 BP（Base Pointer）去指到所要存取的数据，因为 BP 与 SP 寄存器都是相对应 SS（Stack Segment）寄存器的。这个例子显示当间接操作数在堆栈中存取数据时如何利用 BP（作为一基址寄存器）：

```
push bp              ; Save current value of BP
mov bp, sp           ; Set stack frame
push ax              ; Push first; SP=BP-2
push bx              ; push second; SP=BP-4
push cx              ; Push third; SP=BP-6
.
.
.
mov ax, [bp-6]       ; Put third word in AX
mov bx, [bp-4]       ; Put second word in BX
mov bx, [bp-4]       ; Put first word in CX
.
.
.
add sp, 6            ; Restore stack pointer
                     ; (two bytes per push)
pop bp               ; Restore BP
```

如果你经常在程序中使用堆栈值，你可以为它们建立一个标记。例如，你可以使用 TEXTEQU 去建立一个如 count TEXTEQU ‹[bp-6]›的标记。现在你可在先前的实例中使用 mov ax，count 去取代 mov ax，[bp-6] 语句。

4.5.2 储存标志值至堆栈

我们可在程序中使用 PUSHF 与 POPF 指令将标志值 push 或 pop 至堆栈。利用这两个指令可以储存和恢复此标志的状态。你也可以在一子程序中使用它们去储存和恢复调用的程序（caller）或主程序（main）的标志状态。此指令的 32 位版本是 PUSHFD 与 POPFD。

下例子表明在调用 systask 子程序前储存标志寄存器值。

```
pushf
call systask
popf
```

如果你不需要去储存完整的寄存器，你可以使用 LAHF 指令去载入标志寄存器的低 byte 的状态至 AH。SAHF 指令可恢复其值。

4.5.3 存储在堆栈的寄存器值（80186～486Only）

在 80186 处理器可用一个指令（PUSHA 或 POPA 指令）去 push 或 pop 所有通用寄存器。利用这两个指令可在子程序被调用前储存所有通用寄存器的状态，而在 return 后恢复他们。事实上，使用 PUSHA 与 POPA 指令已暗示会较快，且比独自 push 与 pop 每一个寄存器会有较少的程序码。

处理器以下列的顺序 push 这些寄存器（共 8 个）：

AX，CX，DX，BX，SP，BP，SI，DI

而被推入的 SP 值（word）是第一个寄存器被推入之前的 SP 值。处理器是以相反的顺序 pop 这些寄存器。此指令的 32 位版本是 PUSHAD 与 POPAD。

4.6 使用指针与地址存取数据

指针简单来说就是一个包含许多其它变量地址的变量。这个指针中的地址指向其它的对象。当要传过一个在原对象（比如数组）给子程序时，指针是很有用的。主程序（或称调用者；caller）只要放置指针在堆栈中，被调用的子程序就可知道这个数组的位置。这除去必须通过堆栈来回传整个数组的不合实际的步骤。

远程地址和远程指针是不同的。远程地址（far address）是一个配置在远程数据段中的变量地址。远程指针是一个变量，包含段地址和许多其它数据的偏移。像其它的变量一样，指针可以配置在预设的（near）数据段或远程段。

较早版本的 MASM 也有指针变量但是功能不多。在先前的版本，任何地址被载入变量，都可认为指针，如下列的语句：

```
Var     BYTE    0           ; Variable
npVar   WORD    Var         ; Near pointer to variable
fpVar   DWORD   Var         ; Far pointer to variable
```

如果一个变量使用其它变量的名字定义初值（初始化；initialized），这个被初始化的变量就是指针，可以在上面的例子中看到。然而在较早的 MASM 版本，Codeview debugger 认为 npVar 与 fpVar 是 word 和 doubleword 的变量。CodeView 并不认为他们为变量，也不认识他们所指的数据型式（如上例的 BYTE）。

TYPEDEF 伪指令和增强功能的 ASSUME 已能较容易地去管理在寄存器的指针和变量。

4.6.1 使用 TYPEDEF 定义指针变量

TYPEDEF 伪指令可以定义指针变量的类型。这样定义好的类型是被认为是和编译程序本身所提供一样，同样可使用在程序中。当使用去定义指针时，TYPEDEF 文法如下：

typename TYPEDEF [distance] PTR qualifiedtype

typename 是由用户自定类型的新 type 名称。distance 可以是 NEAR 或 FAR。qualifiedtype 可以是由 MASM 本身提供的类型（如 BYTE 或 WORD），或是先前已用 TYPEDEF 定义好的新类型（如下面实例的 PBYTE）。下面有许多用户定义的实例：

```
PBYTE       TYPEDEF         PTR BYTE        ; Pointer to bytes
NPBYTE      TYPEDEF NEAR    PTR BYTE        ; Near pointer to bytes
FPBYTE      TYPEDEF FAR     PTR BYTE        ; Far pointer to bytes
PWORD       TYPEDEF         PTR WORD        ; Pointer to words
NPWORD      TYPEDEF NEAR    PTR WORD        ; Near pointer to words
FPWORD      TYPEDEF FAR     PTR WORD        ; Far pointer to words

PPBYTE      TYPEDEF         PTR PBYTE       ; Pointer to pointer to bytes
                                            ; (in C, an array of strings)
PVOID       TYPEDEF         PTR             ; Pointer to any type of data
```

```
PERSON      STRUCT                              ; Structure type
  name      BYTE  20  DUP  (?)
  num       WORD?
PERSON      ENDS
PPERSON     TYPEDEF   PTR PERSON                ; Pointer to structure type
```

指针的 distance 可由用户自定，或由内存模式（.MODEL 设置）和段大小（16 位或 32 位）自动决定。如果你没有使用 .MODEL，near 指针是缺省值。

在 16 位模式，一个近程指针（near pointer）是 2 bytes；包含这个被指到对象的偏移。一个远程指针（far pointer）需要 4 bytes，包含段和偏移。在 32 位模式，近程指针是 4 bytes，远程指针是 6 bytes，因为在 32 位模式段还是 word（2 bytes）值。如果你用 NEAR 和 FAR 指定 distance，处理器使用目前段大小的缺省 distance。你可以使用 NEAR16、NEAR32、FAR16、FAR32，去推翻由目前段所设置的缺省值。在 FLAT 模式，NEAR 是缺省值。

你可以用由 TYPEDEF 建立的指针类型去说明指针变量。一旦定义好，可以使用在程序中原本所允许的任何地方。下面有许多使用这些指针类型的实例：

```
        ; Type declaration

Array       WORD  25  DUP  (0)
Msg         BYTE      "This is a string", 0
pMsg        PBYTE     Msg                   ; Pointer to string
pArray      PWORD     Array                 ; Pointer to word array
npMsg       NPBYTE    Msg                   ; Near pointer to string
npArray     NPWORD    Arrar                 ; Near pointer to word array
fpArray     FPWORD    Array                 ; Far pointer to word array
fpMsg       FPBYTE    Msg                   ; Far pointer to string

S1          BYTE        "first, 0"          ; Some strings
S2          BYTE        "second", 0
S3          BYTE        "third", 0
pS123       PBYTE     S1, S2, S3, 0         ; Array of pointers to strings
ppS123      PPBYTE    pS123                 ; A pointer to pointers to strings

Andy        PERSON    <>                    ; Structure variable
pAndy       PPERSON   Andy                  ; Pointer to structure variable
                                            ; Procedure prototype

EXTERN      ptrArray:   PBYTE               ; External variable
Sort        PROTO     pArray: PBYTE         ; Parameter for prototype
                                            ; Parameter for procedure
Sort        PROC pArray: PBYTE
```

```
        LOCAL pTmp: PBYTE          ; Local variable
        ...
        ret
Sort    ENDP
```

4.6.2 使用 ASSUME 定义寄存器型式

你可以使用 ASSUME 伪指令和通用寄存器去指定寄存器是一个指针（指到某个大小的对象）。例如：

```
ASSUME      bx: PTR WORD        ; Assume BX is now a word pointer
inc         [bx]                ; Increment word pointed to by BX
add         bx, 2               ; Point to next word
mov         [bx], 0             ; Word Ponted to by BX=0
.
.                               ; Other pointer operations with BX
.
ASSUME      bx: NOTHING         ; Cancel assumption
```

在这个例子，BX 是被指定为一 word 指针。在连续将 BX 当成指针使用之后，通过假设 NOTHING 取消了所有之前的假设。

没有使用 PTR WORD 的假设，许多指令需要一大小的指定。先前的 INC 与 MOV 语句就必须写成像这样去指定操作数的大小：

```
inc    WORD    PTR    [BX]
mov    WORD    PTR    [BX], 0
```

当你已使用了 ASSUME，又尝试去使用这个寄存器做其它的目的，将产生编译错误。例如：

```
; mov al, [bx]          ; Can't mov word to byte register
```

你也可以使用 PTR 运算符去推翻 缺省值：

```
mov al, BYTE PTR [bx]; Legal
```

相同地，你可以使用 ASSUME 去避免将寄存器当成指针，或甚至去 disable 一个寄存器（也就是不能在使用了）：

```
  ASSUME  bx: WORD,     dx: ERROR
; mov     al, [bx]      ; Error-BX is an integer, not a pointer
; mov     al, dx        ; Error-DX disable
```

4.6.3 基本指针和地址运算

程序可以使用指针和地址执行下列基本运算：

(1) 通过储存地址在里面，去初始化一指针变量。

(2) 直接或从指针载入一个地址至寄存器。

下面的实例是假设你已先前使用 TYPEDEF 伪指令定义了下列的指针：

```
PBYTE       TYPEDEF                 BYTE    ; Pointer to bytes
```

```
NPBYTE      TYPEDEF   NEAR    PTR    BYTE   ; Near pointer to bytes
FPBYTE      TYPEDEF    FAR    PTR    BYTE   ; Far pointer to bytes
```

4.6.3.1 初始化指针变量

如果指针的值在编译时间（编译阶段）就知道，编译程序就可自动初始化它，这样在执行时（run time）就不必花时间在这件工作上。下面的实例将告诉你如何将在 Msg 的地址放在指针 pMsg 中。

```
Msg   BYTE    "String", 0
pMsg  PBYTE   Msg
```

如果一个指针变量可以有条件地被定义许多常量地址，要初始化它必定是在执行（run time）时才可完成。在定义近程指针和远程指针初值的技巧是不同的，实例如下：

```
Msg1   BYTE    "String1"
Msg2   BYTE    "String2"
npMsg  NPBYTE  ?
fpMsg  FPBYTE  ?
       .
       .
       .
       mov npMsg, OFFSET Msg1                    ; Load near pointer

       mov WORD PTR fpMsg [0], OFFSET Msg2       ; Load far offset
       mov WORD PTR fpMsg [2], SEG Msg2          ; Load far segment
```

如果你知道远程指针的段值，你可以直接载入它：

```
       mov WORD PTR fpMsg [2], ds                ; Load segment of
                                                 ; far pointer
```

1. 动态寻址

通常一个指针可能需指向一动态地址，意指此地址须依赖执行的条件才能决定。典型的情况包含①内存配置（by MS-DOS）②由 SCAS 与 CMPS 指令所指的地址。下面列出了储存动态地址的技巧：

```
; Dynamically allocated buffer
fpBuf    FPBYTE  0                        ; Initialize so offset will
         .                                ; be zero
         .
         .
         mov  ah, 48h                     ; Allocate memory
         mov  bx, 10h                     ; Request 16 paragraphs
         int  21h                         ; Call DOS
         jc   error                       ; Return segment in AX
         mov  WORD PTR fpBuf [2], ax      ; Load segment
         .                                ; (offset is
```

```
                .                                    ; already 0)
                .
error:                  ; Handle error
```

2. Copying Pointers

有时指针变量必须通过一个寄存器 copy 来初始化。

```
fpBuf1  FPBYTE  ?
fpBuf2  FPBYTE  ?
                .
                .
                .
; Copy through register is faster, but requires a spare register
                mov     ax, WORD PTR fpBuf1 [0]
                mov     WORD PTR fpBuf2 [0], ax
                mov     ax, WORD PTR fpBuf1 [2]
                mov     WORD PTR fpBuf2 [2], ax

; Copy through stack is slower, but does not use a register
                push    WORD PTR fpBuf1 [0]
                push    WORD PTR fpBuf1 [2]
                pop     WORD PTR fpBuf2 [2]
                pop     WORD PTR fpBuf2 [0]
```

3. 指针参数 (Pointers as Arguments)

大部分的高级语言子程序或库函数都是通过堆栈来传递参数，如下所示：

```
; Push a far pointer (segment always pushed first)
                push    WORD PTR fpMsg [2]      ; Push segment
                push    WORD PTR fpMsg [0]      ; Push offset
```

Push 一个地址和 push 一个指到这个地址的指针有相同的结果。

```
; Push a far address as a far pointer
                mov ax, SEG fVar        ; Load and push segment
                push ax
                mov ax, OFFSET fVar     ; Load and push offset
                push ax
```

在 80186 和之后的处理器，可以直接以一个步骤 push 一个常数：

```
                push SEG fVar           ; Push segment
                push OFFSET fVar        ; Push offset
```

4.6.3.2 载入地址至寄存器

在汇编语言程序设计中载入一近程地址至寄存器（或一个远程地址至一组寄存器）是常见的事。为了参考由指针指到的数据，你的程序必须首先放置此指针到一个寄存器或一组寄存器中。载入远程地址需要 segment：offset。下面列出几种使用情况：

Segment：Offset	Standard Use
DS：SI	Source for string operations
ES：DI	Destination for string operations
DS：DX	Input for certain DOS functions
ES：BX	Output from certain DOS functions

1. 数据段寻址

对于一近程地址，你只需要载入偏移；对于堆栈区数据的段假设是SS，而其它数据是DS。对于一个远程指针你必须载入段和偏移。下面实例由近程数据段载入一个地址至DS：BX：

```
        .DATA
Msg     BYTE    "String"
          ...
          mov     bx, OFFSET Msg      ; Load address to BX
                                      ; (DS already loaded)
```

而远程地址可以被载入如下：

```
.FARDATA
Msg     BYTE    "String"
          .
          .
          mov ax, SEG Msg           ; Load address to ES：BX
          mov es, ax
          mov bx, OFFSET Msg
```

你也可以一个步骤从指针读一个远程地址，使用LES和LDS指令即可，描述如下。

2. 远程指针（Far Pointers）

LES与LDS指令载入远程地址至一段组中。这个指令会拷贝指针的低word（2 bytes）至ES或DS寄存器，高word至一指定的寄存器。下面的实例说明了如何载入一个远程指针至ES：DI

```
OutBuf      BYTE    20  DUP   (0)
fpOut       FPBYTE  OutBuf
          ...
          les     di, fpOut     ; Load far pointer into ES：DI
```

3. 堆栈变量

对于要载入一个堆栈变量地址的技巧和载入近程地址的技巧是不同的。对于字符串运算你需要放正确的段值至ES。下面的实例说明了如何载入区段（此为stack）变量的地址至ES：DI：

```
Task PROC
     LOCAL ARG [4]：BYTE

     push ss                ; Since it's stack-based, segment in SS
     pop es                 ; Copy SS to ES
```

```
        lea di, Arg          ; Load offset to DI
```

在这种情况下，这个区段变量事实上是计算至 SS:[BP-4]。这是变量地址堆栈的偏移。因为你不能使用 OFFSET 运算符去得到一个间接操作数的偏移，你必须使用 LEA (Load Effective Address) 指令。

4. 直接操作数

要得到直接操作数的地址，可以使用 LEA 指令或 MOV 指令加 OFFSET。这两种方法有相同的效果，但通过 MOV 指令会产生较小和较快的执行码，实例如下：

```
lea    si, Msg              ; Four byte instruction
mov    si, OFFSET Msg       ; Three byte equivalent
```

5. Copying Between Segment Pairs

要将一组寄存器拷贝给另一组寄存器是相当复杂的，事实上是因为你不能直接拷贝。下面列出两种方法。Timing 是指在 8088。

```
; Copy DS: DI to ES: DI, generating smaller code
        push        ds      ; 1 byte, 14 clocks
        pop         es      ; 1 byte, 12 clocks
        mov         di, si  ; 2 bytes, 2 clocks

; Copy DS: DI to ES: DI, generating faster code
        mov         di, ds  ; 2 bytes, 2 clocks
        mov         es, di  ; 2 bytes, 2 clocks
        mov         di, si  ; 2 bytes, 2 clocks
```

4.6.3.3 独立模式技巧

通常可能要在独立内存模式编写程序。如果你是要编写适用于不同内存模式的程序库，你可以使用条件编译去处理不同大小的指针。你可以使用先前定义的符号@DataSize 和@Model 去测试目前的假设。

先前定义符号@DataSize 测试对于目前内存模式的指针大小：

```
Msg1   BYTE       "String1"
pMsg   PBYTE?
.
.
IF   @DataZize                              ; @DataSize>0 for far
mov WORD PTR pMsg [0], OFFSET Msg1          ; Load far offset
mov WORD PTR pMsg [2], SEG Msg1             ; Load far segment
ELSE                                        ; @DataSize=0 for near
mov pMsg, OFFSET Msg1                       ; Load near pointer
ENDF
```

在下面的实例中，一个子程序接受一参数指针指到一个 word 变量。在子程序中的代码使用@DataSize 去决定目前的内存模式提供的是远程或近程数据：

```
; Procedure that receives an argument by reference
```

```
mu18  PROC  arg: WORD  PTR
      IF    @DataSize
      les   bx, arg          ; Load far pointer to ES: BX
      mov   ax, es [bx]      ; Load thd data pointed to
      ELSE
      mov   bx, arg          ; Load near pointer to BX (assume DS)
      mov   ax, [bx]         ; Load the data pointed to
      ENDIF
      shl ax, 1              ; Multiply by 8
      shl ax, 1
      shl ax, 1
      ret
mu18  ENDP
```

如果你有许多子程序，为了每种情况而编写条件编译，将是很烦人的。下列的条件语句自动产生适合的指令与段指定。

```
; Equates for conditional handling of pointer
        IF @DataSize
lesIF   TEXTEQU  <les>
ldsIF   TEXTEQU  <lds>
 esIF   TEXTEQU  <es: >
        ELSE
lesIF   TEXTEQU  <mov>
ldsIF   TEXTEQU  <mov>
esIF    TEXTEQU  <    >
          ENDIF
```

一旦你定义了这些条件，你可以简化程序必须处理许多指针类型的工作。下列的实例使用了条件程序码重写了上述的 mu18 子程序：

```
mu18      PROC  arg: PTR WORD
          lesIF   bx, arg          ; Load pointer to BX or ES: BX
          mov     ax, esIF [bx]    ; Load the data from [BX] or ES: [BX]
          sh1     ax, 1            ; Multiply by 8
          sh1     ax, 1
          sh1     ax, 1
          ret
mu18      ENDP
```

利用这些条件语句可以只定义一次后放在包含文件中 (include file)，无论你何时需要去处理指针都可以使用。

第5章　说明与使用数据类型

编译程序将原始程序产生目的模块。此目的模块包含许多机器语言，它也包含在程序中将要使用数据的存储空间。在程序中你必须为每个你要使用的数据配置空间。你可以借助使用定义数据伪指令去指定编译程序。此伪指令可以定义 bytes、words、doublewords、quadwords 和甚至更大的数据项目。

编译程序会一列一列地编译程序，当有定义数据伪指令出现时，编译程序会为此伪指令配置所需要的内存空间，通常是在数据或额外段中定义。

这章节内容包括在程序中使用数据类型的重要概念。第 1 节说明如何说明简单的数据类型——整型变量。第 2 节描述复杂的数据类型——array、string、record、structure 和 union。

5.1　定义与使用简单数据类型

5.1.1　说明整型变量

你可以使用数据配置伪指令去定义整型变量的初值。这章节将解释如何在程序中使用 SIZEOF 和 TYPE 运算符去告诉编译程序有关此类型的信息。

5.1.2　配置整型变量的内存空间

当您通过指派一个标记给数据配置伪指令说明一个整型变量时，编译程序会为这个整数配置内存空间。变量名称将变成一个内存空间的标记，以使指令能存取这个变量。文法如下：

[name]　　directive　　initializer

name 是选择性可有可无的，如果有，指令可借此名称存取此变量。directive 是数据配置伪指令，表 5-1 列出所有可用的数据配置伪指令，不同的数据配置伪指令各有不同的属性。initializer 是代表初值，可以是一个常数（整数）、一连串的常数、一个常数运算式。你可以使用数据伪指令以一个步骤即可同时说明与定义变量初值。实例如下：

```
integer         BYTE        16              ; Initialize byte to 16
negint          SBYTE       -10             ; Initialize signed byte to -16
expression      WORD        2 * 5           ; Initialize word to 10
signedexp       SWORD       5 * 3           ; Initialize signed word 15
empty           QWORD       ?               ; Allocate uninitialized long int
                BYTE        1, 2, 3, 4, 5   ; Initialize six unnamed bytes
long            DWORD       4294967295      ; Initialize doubleword to
                                            ; 4, 294, 967, 295
longnum         SDWORD      -2147433648     ; Initialize signed doubleword
                                            ; to -2, 147, 433, 648
tb              TBYTE       2345t           ; Initialize 10-byte binary number
```

下列两个语句是相同的，DB（define byte）是旧式的写法，BYTE 是新式的写法，详细

的内容说明在表 5-1：

```
DATA1    DB      12H
DATA1    BYTE    12H
```

初值必须符合变量说明时的大小，具有初值的变量一般是放在 .DATA 段，若变量定义时不指定初值，可在定义初值的地方使用※？※如下：

```
    DATA2    WORD    ?
;   DATA2    BYTE  1234H        ; illegal（不合法）
```

此不具初值的变量一般是放在 .DATA? 段，当然也可以放在 .DATA 段。若要考虑与 Microsoft 高级语言连结，建议最好还是将不具初值的变量存放在 .DATA? 数据段。虽然也可以把数据存放在代码段，并在开头处加上一个标记，如下：

```
.CODE
    mov ax, @code       ; 因 data _ address 标记在 .CODE 程序
    mov ds, ax          ; 段，所以在存取之前，必须先将
                        ; ds 指向 .CODE 代码段
data _ address;
    byte 'Hello, how are you?'
```

不过我们不鼓励这样使用，这样容易造成程序与数据的混淆，不易管理。

下列在表 5-1 的数据伪指令说明此整数的大小及值的范围：

表 5-1　数据伪指令

指　令	说　明
BYTE, DB (byte)	配置无符号数从 0 到 255
SBYTE (signed byte)	配置有符号数从－128 到＋127
WORD, DW (word＝2 bytes)	配置无符号数从 0 到 65，535 (64K)
SWORD (signed word)	配置有符号数从－32，768 到＋32，767
DWORD, DD (doubleword＝4 bytes)	配置无符号数从 0 到 4，294，967，295 (4MB)
SDWORD (signed doubleword)	配置有符号数从－2，147，483，648 到＋2，147，483，647
FWORD, DF (farword＝6 bytes)	配置 6－byte (48 位) 整数。在 80386/486 处理器正常是只使用作为指针变量
QWORD, DQ (quadword＝8 bytes)	配置 8－byte 整数使用在 8087 家族协处理器指令
TBYTE, DT (10 bytes)	配置 10－byte (80 位) 整数

编译程序储存整数是将最低的字节 (byte) 储存在最低的内存。注意编译程序列表和大部分的纠错器 (debugger) 是以一个相反的顺序来表示一个 WORD，也就是先列出较高的位组。图 5-1 列出这些整数格式。

虽然 TYPEDEF 伪指令主要的目的是定义指针变量，你也可以使用 TYPEDEF 去建立一个任何整数类型的别名。例如，这些说明：

```
char    TYPEDEF  SBYTE
```

long　TYPEDEF　DWORD
float　TYPEDEF　REAL4
double　TYPEDEF　REAL8

允许你去使用 char、long、float 或 double 在你的程序中，就如同 C 的用户自定类型，如果你喜欢 C 的数据类型。

注：REAL4、REAL8 和 REAL10 伪指令可配置实数。

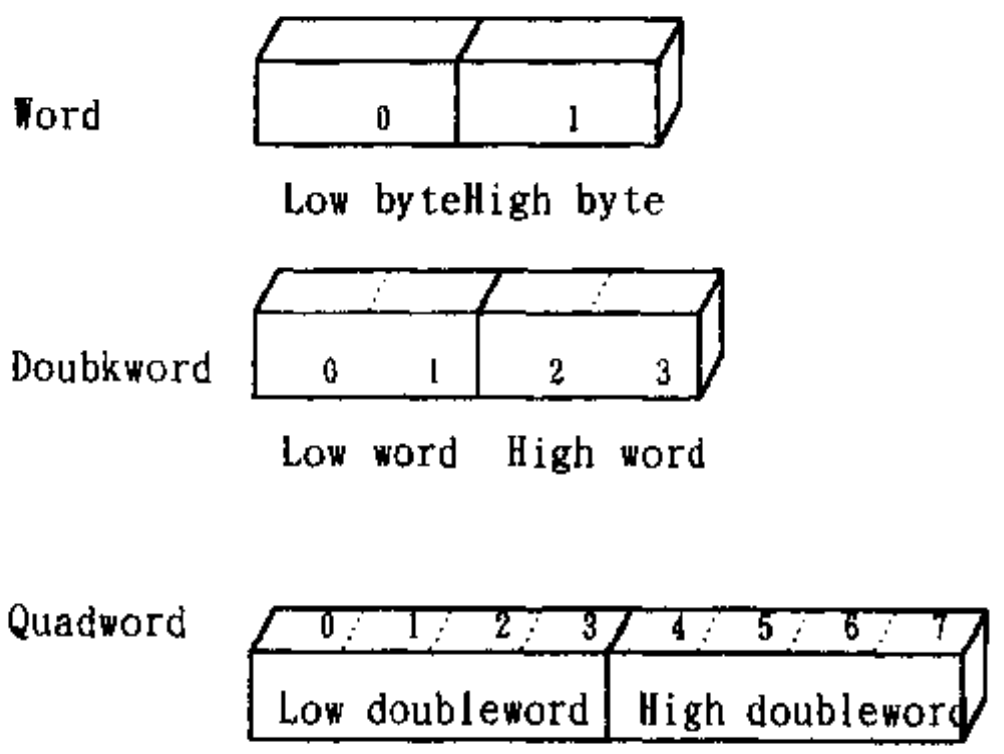


图 5-1　整数格式

5.1.3　数据初值

你可以在数据说明时就用常数或一个可计算成常数的运算式去定义初值。如果你指定的初值太大，编译程序会产生一个 error 信息。如果你不需要编译程序去定义变量的初值，可以放"?"在定义初值(initializer)的地方表示未具初值，编译程序会配置这个空间但不会写入值。也就是使用"?"，代表需在运行才可初始化变量。你可以用数据伪指令(如 BYTE；DB 或 WORD；DW)，在说明变量时就顺便定义初值，一个步骤就可完成。

5.1.4　使用简单变量

因为 MASM 指令需要操作数有相同的大小，你可以使用 PTR 运算符去指定操作数的大小，而不管说明时的大小，相当灵活，但很危险。例如你可以使用 PTR 运算符去存取一个 DWORD 大小变量的较高 word。PTR 运算符的文法如下：

type　PTR　表达式

PTR 运算符可强迫表达式拥有与用户指定有相同的 type。如下：

```
        .DATA
num   DWORD   0
        .CODE
        mov ax, WORD PTR num [0] ; Loads a word-size value from
        mov dx, WORD PTR num [2] ; a doubleword variable
```

5.2　定义和使用复杂数据类型

在 MASM 6.11 中复杂数据类型变量有——arrays、strings、records、structures 和 unions。你可以存取数据如一个单元或一个独立元素。而复杂数据类型的独立元素通常是整数。

5.2.1　字符串和数组

数组是一连续数的集合物，所有的变量具有相同的大小和数据类型，又称为元素。而字符串就是一个字符数组。例如，字符串"ABC"，每个字符就是一个元素。你可以通过数组或字符串的第一个元素去存取其它的元素。

1. 说明和存取数组元素

数组的元素占据连续的内存，这样程序可以通过数组的开关去参考每个元素。为了要说明一个数组，必须提供一个标记名称（即数组名称）、元素类型和一连串的起始值或"?"。下列的实例说明了 warray 和 qarray 数组：

```
warray      WORD      1，2，3，4
qarray      QWORD     OFFFFFFFFH，12345678H
```

数组的初值可以跨越多列说明。不过第一个初值必须要和数据类型说明在同一列，且所有的元素都需要初值。如果有跳列，则需在每列结尾加上"'"。下列实例都是合法的数组说明：

```
big         BYTE      21，22，23，24，25，26，27，28
somelist WORD   10，
                20，
                30
```

注：注意最后一个元素后面不须"，"

如果你没有使用 LENGTHOF 和 SIZEOF 运算符，你也可以使用下列的方式说明数组。注意每列的开头都需有类型说明：

```
var1      BYTE      10，20，30
          BYTE      40，50，60
          BYTE      70，80，90
```

注：同样是储存在连续 9 bytes 的连续内存空间。

2. DUP 运算符

你也可以使用 DUP 运算符说明数组。文法如下：

```
count     DUP       （初值［，初值］……）
```

count 值是重复括号的值多少次的意思。而初值可以是整数、字符常数（如'A'）或另一个 DUP 运算符，不过仍然必须在括号内。例如，这叙述

```
barray      BYTE      5      DUP       (1)
```

会配置整数 1 共五次，所以共 5 bytes。

下面的实例列出许多使用 DUP 运算符去配置数据的方式：

```
array          DWORD     10        DUP (1)                    ; 10 doublewords
                                                              ;      initialized to 1
buffer         BYTE      256       DUP (?)                    ; 256-byte buffer
masks          BYTE      20        DUP (40h, 20h, 04h, 02h)   ; 80-byte buffer
                                                              ;    with bit masks
three _d       DWORD     5         DUP (5 DUP (5 DUP (0) ) )  ; 125 doublewords
                                                              ; initialized to 0
```

3. 存取数组元素

要存取数值中的每个元素可利用一索引值，当然是从 0 开始。而数组的索引值是放在数组名字之后的中括号内，如：array［10］

汇编语言的索引值和高级语言的索引值意义不同。汇编语言的索引值是代表元素的偏移即位置，而不是第几个元素的意思。例如在 C，array［10］是代表参考数组的第 11 个元素，

而不论元素的大小是 1 byte 或 2 bytes。在汇编语言如果索引值是参考一个元素大小为 byte 的数组，则索引值和元素的位置正好是相同不必有其它参考量，例如定义一个数组

```
prime    BYTE    1，3，5，7，9，11，13，17
```

prime [0] 的值是 1，prime [1] 的值是 3，以此类推。

然而如果数组元素的大小超过 1 byte，索引值数目(除了 0 以外)并不和元素的位置相符。你必须将元素的位置乘上元素的大小，放至中括弧内才能正确存取你要的数据。因此对于这个数组

```
wprime    WORD    1，3，5，7，9，11，13，17
```

wprime [0] 是表示第一个元素 1，wprime [2] 是表示第二个元素 3。而 wprime [4] 是表示第三个元素 5，也就是由数组开头地址算起的 4 bytes 之处。下列实例的索引值是决定在运行时。sh1 si，1 是代表向左移一位（即乘 2 之意；因为元素的大小是 word)：

```
mov     si,    cx              ; CX holds position number
sh1     si,    1               ; Scale for word referencing
mov     ax,    wprime [si]     ; Move element into AX
```

要存取一个数组元素偏移，可以下列的公式计算：

nth element of array= [array (n－1) * size of element]

因此如果数组 wprime 开始在地址 DS：2400H，代表处理器要存取在 DS：2406H 的偏移。

你也可以利用"＋"运算符去代表"[]"，下面的两种方式是相同的。

```
wprime [10]
wprime+10
```

中括号的意义主要是要加上偏移地址，若你只要存取第一个元素，那你就不需要使用"[]"。因此，wprime 和 wprime [0] 同样是代表要存取 wprime 数组第一个元素。

4. LENGTHOF，SIZEOF 和 TYPE 运算符

在数组的应用中，LENGTHOF、SIZEOF 和 TYPE 运算符可返回有关数组的长度，大小的信息和初值的类型。

LENGTHOF 运算符可返回数组的元素数目。SIZEOF 运算符可返回在数组定义时的初值有多少 byte 数。TYPE 返回数组元素的大小。

```
array    WORD    40  DUP   (5)

larray   EQU     LENGTHOF     array      ; 40 elements
sarray   EQU     SIZEOF       array      ; 80 bytes
tarray   EQU     TYPE         array      ; 2 bytes per elements
num      DWORD   4, 5, 6, 7, 8, 9, 10, 11

lnum     EQU     LENGTHOF     num    ; 8 element
snum     EQU     SIZEOF       num    ; 32 bytes
tnum     EQU     TYPE         num    ; 4 bytes per element
```

```
warray          WORD        40    DUP (40 DUP (5))

len     EQU     LENGTHOF    warray    ; 1600 elements
siz     EQU     SIZEOF      warray    ; 3200 bytes
typ     EQU     TYPE        warray    ; 2 bytes per elements
```

5. 说明和初始化字符串

字符串是一个字符数组。比如要定义字符串的初值为"Hello，there"，对于字符串中的每个字符会配置 1 byte 的内存空间。字符串在定义初值时不可定义超过 255 个字符。

如果数据伪指令不是 BYTE，字符串在定义初值时只能定义第一个元素，且初值的大小必须和指定的大小一样。

```
wstr        WORD    "OK"
dwstr       DWORD   "DATA"      ; legal under EXPR32 only
```

和数组一样，字符串定义初值也可分散在许多列，而每列的结尾需有一个“,”（如果超过一列的话）：

```
str         BYTE        "This is a long string that does not",
                        "fit on one line."
msg         BYTE        "This string extends",
                        "over there",
                        "lines."

lmsg  EQU   LENGTHOF    msg   ; 37 elements
smsg  EQU   SIZEOF      msg   ; 37 bytes
tmsg  EQU   TYPE        msg   ; 1 byte per element.
```

5.2.2 结构与联合 (structure and union)

结构可能是不相同的数据类型和可当成一个单元或可存取其任何组成成份的变量的一个集合。在结构中的区段可以有不同的大小和数据类型。

联合和结构是相同的（除了联合中的区段重叠在相同的内存）。联合允许你在相同的内存空间定义不同的数据格式。也就是在不同的情况下可存储不同的数据类型。它也可以将存储的数据视为一种数据类型稍后又恢复为另一种数据类型。而在结构中任何地方的区段都有一个相对应结构开头第一个 byte 的偏移，而在联合中的所有区段均开始在相同的偏移。结构的大小是所有成份的总和；联合的大小是最长区段的长度。

学过 C 语言的读者应当相当熟悉，MASM 的结构是和 C 的 struct、FORTRAN 的 STRUCTURE、Pascal 的 RECORD 相似。而联合就和 C 的 union 相似。

使用结构和联合需遵循下列的步骤：

①说明结构（或联合）的类型。

②定义一个或更多此类型的变量。

③直接或间接使用区段运算符“.”存取其中的区段。

在汇编语言叙述中，你可以使用完整的结构或联合变量或只是将其中的独立区段当成操作数。

1. 说明结构与联合类型

当你在说明一个结构或联合时，你可以为数据建立一个样本。这个样本陈述了结构或联合的大小、可选择项目、初值，但并不配置内存。

STRUCT 和 UNION 类型说明的格式如下：

```
name    {STRUCT|UNION} [alignment] [, NONUNIQUE]
        区段初值
name    ENDS
```

区段初值是一连串一个或多个变量说明。你可以说明缺省的初值或使用 DUP 运算符。

(1) 定义区段初值

当你说明结构或联合的区段类型时提供了区段初值，当你定义这个类型的变量时，这些初值将变成这个区段的缺省值。

当你定义一个联合类型的区段初值时，这第一个区段的类型和初值将变成这个联合的缺省类型和初值。在这个已有初值的联合说明实例中，这个联合的缺省类型就是 DWORD，缺省初值为 OFFh：

```
DWB   UNION
    d   DWORD     00FFh
    w   WORD      ?
    b   BYTE      ?
DWB   ENDS
```

如果第一个数字的大小小于联合的大小，编译程序会将联合的其余部份的初值设为 0。当字符串在定义初值时，在确定字符串的初值不要超过最大的可能值。

(2) 区段名称

结构和联合中同一层的区段名称必须是唯一的，因为他们是表示区段从结构开关算起的偏移。

在程序中任何地方的标记可以有和结构区段相同的名称，但不包括宏。一个已说明过的区段名称可以再出现在其它的结构中，不需要是唯一的。如果你放置 OPTION M510 或 OPTION OLDSTRUCTS 在程序中或在命令行中使用/Zm 参数，区段名称必须是唯一的，因为 MASM 6.0 版之前区段名称必须是唯一的。

(3) 结构的边界 (alignment) 值和偏移

在有定义起始边界值的区段比没有定义起始边界区段进行存取结构的数据是较快的。因此，起始边界的定义换取速度浪费了空间。程序执行在 16 位或 32 位处理器 alignment 可增进存取，但在 8 位 8088 处理器却没有差别。

编译程序定义结构区段的方式决定了需要去储存变量的所有空间。在结构中的每个区段都有一个和 0 有关的偏移。如果你在结构说明时指定 alignment 或在命令行使用/Zpn 参数，对于每个区段的偏移可以被 alignment (或 n) 修改。

可被接受的 alignment 值是 1，2，4，缺省值是 1。如果类型说明时包含了 alignament，每个区段定义的边界不是区段的大小就是 alignment 值。如果区段的大小大于 alignment 值，区段将被填塞，这样它的偏移才可被 alignment 值整除，否则，区段将被填塞使得偏移可被区段大小整除。

对于区段为了载入时能到达正确的偏移所需要的填塞值，都是增加在将配置的区段之前，而且所填塞的值总是0。最后结构的大小也必须被结构的 alignement 值整除，所以 0 也可以被增加在结构的结尾。

如果没有 alignment 或在命令行中使用参数/Zp，偏移是按照每个数据伪指令的大小增加。这和缺省时 alignment＝1 相同。在类型说明时指定 alignment 会推翻在命令行中使用的/Zp参数。

这个实例说明编译程序如何决定偏移：

```
STUDENT2    STRUCT    2          ; Alignment value is 2
  score     WORD      1          ; Offset＝0
  id        BYTE      2          ; Offset＝2 (1 byte padding added)
  year      DWORD     3          ; Offset＝4
  sname     BYTE      4          ; Offset＝8 (1 byte padding added)
STUDENT2 ENDS
```

1 byte 的填塞值被增加在第一个 byte 大小的区段结尾。否则 year 区段的偏移应该是 3（是不能被 alignment 值 2 整除；因为 dword＞alignment 2，所以应考虑 alignment 2）。又现在结构的大小是 9 bytes，因为 9 不能被 2 整除，所以有 1 byte 的填塞值在 Student2 的结尾。起始边界值影响结构变量的起始边界，所以增加起始边界值影响内存的使用。这个特征提供了与 Microsoft C 结构的兼容性。

```
STUDENT4    STRUCT    4              ; Alignment value is 4
  sname     BYTE      1              ; Offset＝0 (1 byte padding
                                         added)
  score     WORD      10 DUP (100)   ; Offset＝2
  year      BYTE      2              ; Offset＝22 (1 byte padding
                                         added so offset of next
                                         field is divisible by 4)
  id        DWORD     3              ; Offset＝24
STUDENT4    ENDS
```

ALIGN、EVEN、ORG 伪指令可以修改在结构定义时区段偏移如何被放置。EVEN 和 ALIGN 伪指令插入填塞的 bytes 至区段偏移附近至指定的起始边界。ORG 伪指令改变下一个区段的偏移至一个指定值（可为正数或负数）。当说明结构时，如果你使用 ORG，你不可以再定义一个此类型的结构。当存取存在数据的结构时，ORG 是很有用的，比如一个由高级语言建立的堆栈页框（stack frame）。

2. 定义结构与联合变量

一旦你已说明了结构或联合类型，你可以定义此类型的变量。对于已定义好的变量，内存是配置在类型格式被说明的段。对于定义结构或联合变量的语法是：

[name] typename ＜ [初值 [，初值] ....]＞

[name] typename { [初值 [，初值] ....]}

[name] typename constant DUP ({ [初值] [，初值] ....])

name 是被指派给变量的标记。如果你没有提供一个名称，编译程序会为这个变量配置空间，但不会给它一个符号名称。typename 是之前被说明的结构或联合类型名称。

你可以为每个区段指定初值。每个初值必须和类型说明所定义的区段类型一致。对于联合来说，初值的类型必须和第一个区段的类型一致。一连串的初值也可以使用 DUP 运算符。

一连串的初值中间用","隔开，或在每列结尾使用"\"。最后一个大括号"}"或">"必须和最后一个初值在同一列。你也可以使用"\"去延伸一列，如下例 Item4 的说明。而"{"与"<"括号也可以混用在初值中，只要类型相配即可。下列实例列出了定义一连串 ITEMS 类型结构的选项。

```
ITEMS           STRUCT
    Iname       BYTE        'Item Name'
    Inum        WORD        ?
    UNION       ITYPE                       ; UNION keyword appears first
        oldtype BYTE 0                      ; when nested in structure,
        newtype WORD ?                      ; ( See "Nested structures
    ENDS                                    ;       and Unions, "following) .
ITEMS           ENDS
            .
            .DATA
Item1       ITEMS   <   >                   ; Accepts default initializers
Item2       ITEMS           {   }           ; Accepts default initializers
Item3       ITEMS           <'Bolts', 126>  ; Override default value of first
                                            ; 2 fields ; use default of
                                            ;   the third field
Item4       ITEMS   {\
                            'Bolts',        ; Item name
                             126\           ; Part number
                            }
```

※ 如果没有初值定义，还是需要"{ }"或"< >"。

你不需要定义结构中所有区段的初值。如果区段是空的，编译程序会使用结构区段说明的初值。如果没有缺省值，则区段值将是一个未定值。

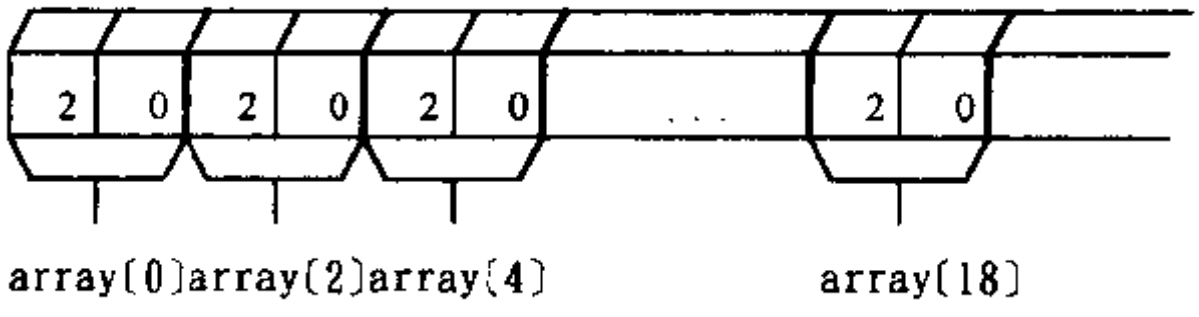
然而对于网状结构或联合，这些是相等的：

```
Item5       ITEMS       {'Bolt',,       }
Item6       ITEMS       {'Bolt',, {   }}
```

联合类型 WB 可以看成一个变量或数组：

```
WB  UNION
  w WORD        ?
  b BYTE        ?
WB  ENDS
```

```
num     WB      {0FH}                          ; Store 0FH
array   WB      (40 / SIZEOF WB) DUP ({2})     ; Allocates and
                                               ; initializes 20 union
```



(1) 数组为区段初值

初值的大小决定数组的长度，数组可以推翻在变量定义时区段的内容。而数组所新定义的数据个数不可超过缺省值。如果数组定义了 n 个元素且小于缺省值，则只会改变前面的 n 个值，数组其余未指定的元素将以原始的初值填入。

(2) 字符串为区段初值

如果新指定的字符串比缺省的长度还短，编译程序会将其长度设成和缺省的长度一样，而以空白填入。如果缺省值是一个字符串，而指定的值不是字符串，那么指定的值必须以"{ }"或"< >"围起来。

字符串可以强定任意数目的 BYTE（或 SBYTE）类型元素。你不需要在"{ }"或"< >"内将字符串围起来，除非和其它的强定方法混用在一起。

如果结构有一个区段初值是字符串或是元素为 BYTE 的数组，任何新的字符串被指派给变量的区段若小于缺省值，将用空白填充。编译程序会增加四个空白在前面定义的 ITEMS 类型变量的'Bolts'结尾处。在 ITEMS 结构的 Iname 区段不可以包含比'Item Name'还长的区段初值。

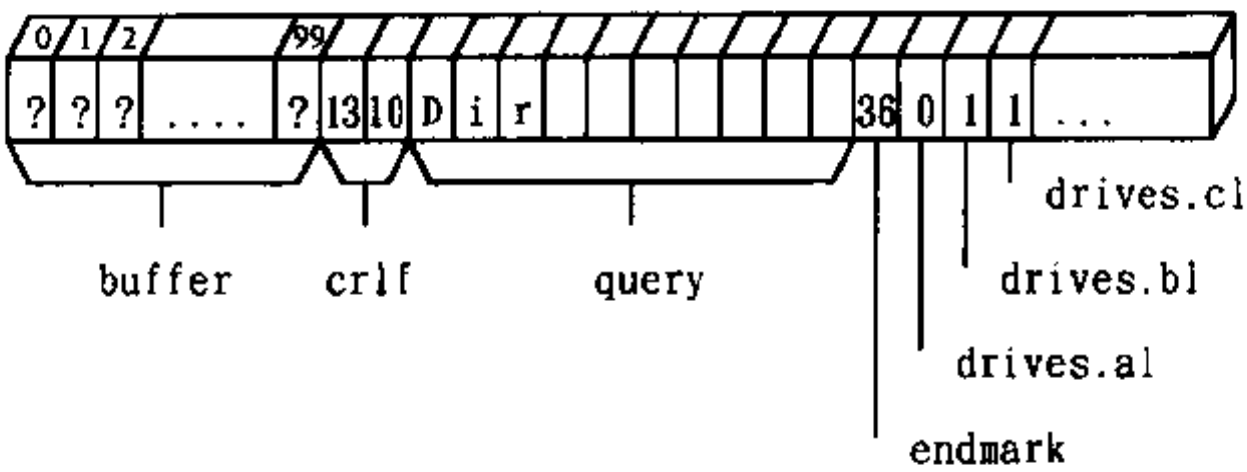
(3) 结构为区段初值

对于结构变量的初值必须以"{ }"或"< >"围起来，但是你可以指定比缺省值还少的元素。这个例子列出使用结构当成区段初值的缺省值使用实例：

```
DISKDRIVES    STRUCT
      al        BYTE    ?
      bl        BYTE    ?
      cl        BYTE    ?
DISKDRIVES    ENDS
INFO          STRUCT
  buffer        BYTE  100 DUP (?)
  crlf          BYTE  13, 10
  query         BYTE  'Filename:'          ; String (=can override
  endmark       BYTE  36
  drives        DISKDRIVES (0, 1, 1)
INFO          ENDS
```

```
infol           INFO    {,    ,    'Dir'}
; Next line illegal since name in query field is too long:
; info2         INFO    { "TESTFILE",    , "DirectoryName"}
lotsof          INFO    {,    , 'file1',    , {0, 0, 0}},
                        {,    , 'file2',    , {0, 0, 1}},
                        {,    , 'file3',    , {0, 0, 2}}
```

下列的图形说明编译程序如何储存 info1：



drives 的初值指定了对于这个结构所有三个区段的缺省值。在 info1 的空白栏位使用这些区段的缺省值。而 info2 的说明是不合法的，因为“DirectoryName”比那个区段的起始字符串还长。

(4) 结构中定义数组

你可以使用 DUP 运算符定义一个结构数组或通过它建立一连串的结构。例如，你可以定义一个结构数组变量，如下：

```
Item7     ITEMS     30     DUP ( {,, {10}})
```

这个 Item7 数组定义有 30 个 ITEMS 类型的元素，每个元素的第三个区段初值都设为 10。你也可以列出数组元素如下实例所示：

```
Item8     ITEMS     {'Bolts', 126, 10},
                    {'Pliers', 139, 10},
                    {'Saws', 414, 10},
```

(5) 结构中的 LENGTHOF、SIZEOF 和 TYPE

结构的大小可由 SIZEOF 得知，或由最后一个区段的偏移加上最后一个区段的大小，再加上为了适当的边界值所需要加入的任何填塞值所决定。此实例使用先前的数据来说明在结构中使用 LENGTHOF、SIZEOF 和 TYPE 运算符。

```
INFO      STRUCT
    buffer      BYTE      100   DUP (?)
```

```
INFO          STRUCT
  buffer      BYTE        100 DUP (?)
  crlf        BYTE        13, 10
  query       BYTE        'Filename:'
  endmark     BYTE        36
  drives      DISKDRIVES <0, 1, 1>
```

```
INFO        ENDS

infol       INFO        {,    , 'Dir'}
lotsof      INFO        {,    , 'file1',    , {0, 0, 0}},
                        {,    , 'file2',    , {0, 0, 1}},
                        {,    , 'file3',    , {0, 0, 2}}

sinfol      EQU         SIZEOF          infol       ; 116=number of bytes in
                                                    ; initializers
linfol      EQU         LENGTHOF        infol       ; 1=number of items
tinfol      EQU         TYPE            infol       ; 116=same as size

slotsof     EQU         SIZEOF          lotsof      ; 116 * 3=number of bytes in
                                                    ; initializers
llotsof     EQU         LENGTHOF        lotsof      ; 3=number of items
tlotsof     EQU         TYPE            lotsof      ; 116=same as size for structure
                                                    ; of type INFO
```

(6) 联合中的 LENGTHOF、SIZEOF 和 TYPE

联合的大小可由 SIZEOF 得知，或由最长的区段加上所需的填塞值所决定。联合变量的长度(即项数)可由 LENGTHOF 得知，这和定义在"{ }"或"< >"内的初值数目相等。TYPE 会返回此类型最长区段的值。

```
DWB         UNION
  d         DWORD       ?
  w         WORD        ?
  b         BYTE        ?
DWB         ENDS

 num        DWB         {OFFFFh}
 array      DWB         (100/SIZEOF DWB) DUP ( {0})

 snum       EQU         SIZEOF          num     ; =4
 lnum       EQU         LENGTHOF        num     ; =1
 tnum       EQU         TYPE            num     ; =4
 sarray     EQU         SIZEOF          array   ; =100 (4 * 25)
 larray     EQU         LENGTHOF        array   ; =25
 tarray     EQU         TYPE            array   ; =4
```

3. 存取结构、联合和区段

像其它变量一样，结构变量也可以通过名字存取。你可以使用这个文件去存取结构变量中的区段：

variable. field

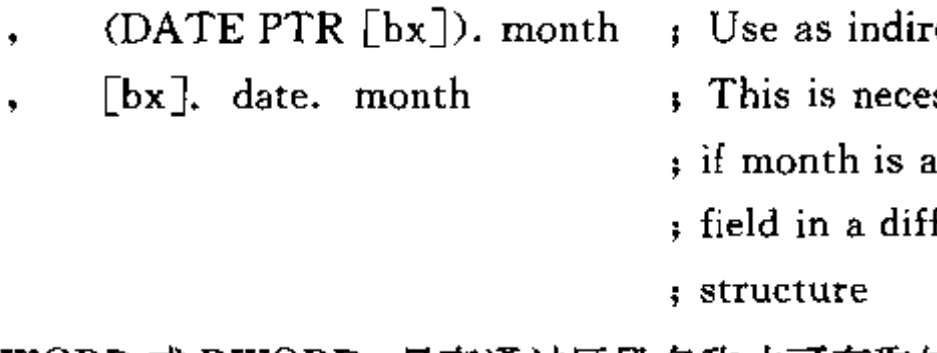
使用结构或联合名字与在区段名字之前使用点运算符 (.)，就可存取所有的区段。编译程序只有在使用结构区段时才需要你使用点运算符。下一实例说明去存取类型 DATE 结构区

段的许多方式：

```
DATE        STRUCT                          ; Defines structure type
 month      BYTE      ?
 day        BYTE      ?
 year       WORD      ?
DATE        ENDS

yesterday     DATE          {1, 20, 1995}   ; Declare structure
                                            ; variable
        .
        .
mov         al,    yesterday. day           ; Use structure variables
mov         bx;    Offset yesterday         ; Load structure address
mov         al,    (DATE PTR [bx]). month   ; Use as indirect operand
mov         al,    [bx]. date. month        ; This is necessary only
                                            ; if month is already a
                                            ; field in a different
                                            ; structure
```

若联合的大小是 WORD 或 DWORD，只有通过区段名称才可存取结构或联合。在下面的实例，两个 MOV 语句说明如何存取联合数组的元素：

```
WB  UNION
 w  WORD        ?
 b  BYTE        ?
WB  ENDS

array  WB       (100/SIZEOF WB) DUP ( {0})

       mov      array [12] . w, 40h
mov             array [32] . b, 2
```



如前面的程序所列，你可以使用联合去存取相同的数据在一个形式以上。结构和联合的应用是为了要简化重定位一个还程指针的工作。一个还程指针说明如下：

```
FPWORD     TYPEDEF     FAR   PTR   WORD
```

```
        .DATA
WordPtr    FPWORD  ?
```

为了要将 WordPtr 指到在目前数据段一个大小为 word 的 ThisWord，你必须遵循下面的步骤：

```
mov    WORD    PTR    WordPtr [2],    ds
mov    WORD    PTR    WordPtr, OFFSET ThisWord
```

前面的方法需要你记住段或偏移要先载入。然而，如果你的程序说明一个联合如下：

```
uptr    UNION
  dwptr    FPWORD  0
  STRUCT
    offs    WORD    0
    segm    WORD    0
  ENDS
  uptr      ENDS
```

你可以使用下面的步骤去初始化还程指针：

```
        .DATA
WrdPtr2  uptr   〈〉
         .
         .
         .
         mov    WrdPtr2.segm, ds
         mov    WrdPtr2.offs, OFFSET ThisWord
```

这段程序搬移段和偏移到指针，然后使用联合其它的区段搬移指针到寄存器。虽然这个技巧不会减少程序码的大小，但是它可避免对于载入段和偏移的顺序所造成的困惑。

4. 网状结构与联合

你可以使用许多方式去网状化结构和联合。这一节将解释如何去存取在网状结构或联合的区段。下列的实例列出网状化的四种技巧以及如何去存取区段。

注意对于网状结构的文法。

```
ITEMS          STRUCT
  Inum         WORD        ?
  Iname        BYTE        'Item Name'
ITEMS          ENDS

INVENTORY      STRUCT
    UpDate     WORD        ?
    oldItem    ITEMS       {\
                           100,
                           'AF8' \          ; Name variable of existing
                           }                ; structure
               ITEMS       {mov yearly.     ; Unnamed variable of
```

```
                                    Inum, 'C', '94C' }
                                                              ; existing type
STRUCT          ups                                       ; Named nested structure

  source          WORD          ?
  shipmode        BYTE          ?
ENDS
    STRUCT                                                ; Unnamed nested structure
      f1          WORD          ?
      f2          WORD          ?
    ENDS
INVENTORY         ENDS

                  .DATA
yearly            INVENTORY     {}

; Referencing each type of data in the yearly strcture:

                  mov           ax, yearly.oldItem.Inum
                  mov           yearly.ups.shipmode, 'A'
                  mov           yearly.Inum, 'C'
                  mov           ax, yearly.f1
```

对于网状结构与联合，你可以使用这些技巧的任何一种：

◆结构或联合的区段可以说明一个已存在的结构或联合类型的变量，如 oldItem 区段。因为 INVENTORY 包含两个类型为 ITEMS 的结构，而在 oldItem 的区段名称不是唯一的。因此，当你要存取这些区段时，你必须写上完整的区段名称，如语句：

mov ax, yearly. oldItem. Inum

◆为了要说明一个已命名的结构或联合在另一个结构或联合中，必须先写 STRUCT 或 UNION 这两个保留字，然后再为它定义一个标记（即名称）。对于网状结构或联合区段的存取，总是如下所示：

mov yearly. ups. shipmode, 'A'

◆如说明在 INVENTORY 的 Items 区段，你可以使用已存在的结构或联合的未命名的变量在另一个结构或联合中。在这些情况下，你可以直接存取其中的区段：

mov yearly. Inum, 'C'; '94C' 之前的区段

mov ax, yearly. f1

5.2.3 记录（Record）

记录和结构是相似的（除了在记录中的区段是位字符串；bit string）。在记录中的每位区段可分别以常数操作数或运算式使用。处理器不能在执行时处理个别的位，但可使用操纵位的指令存取位区段。

一般来说，对于使用记录变量的三个步骤和使用其它复杂数据类型是相同的：

1. 说明记录类型。
2. 定义一个或更多有记录类型的变量。
3. 使用移位（shift）或标签（mask）存取记录变量。

一旦已定义好，你可以在汇编语言中使用记录变量作为操作数。这章节解释记录说明文法和使用 MASK 与 WIDTH 运算符。

4. 说明记录类型

在编译时不会为记录配置内存空间，而是在程序执行时来完成的。RECORD 伪指令可说明一个 8 Bit、16 Bit 或 32 Bit 记录（包含一个或多位区段）的记录类型。语法是：

recordname　　RECORD　　field　　[，field] ....

field 说明了区段的名称、宽度和初值。每一个 field 的语法是：

fieldname：width [=expression]

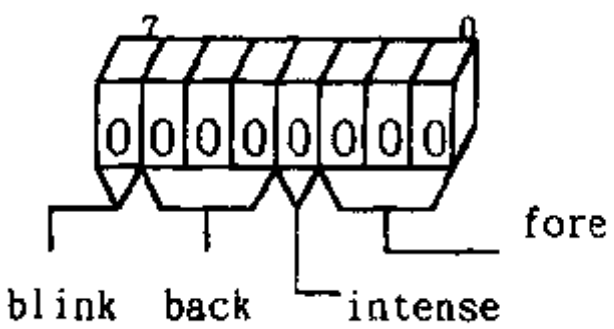
全域的（可视为整个程序）标记、宏名称和记录区段名称必须是唯一的，但记录区段名称可以和结构区段名称相同。Width 是在此区段内的位数。expression 是一个常数，作为此区段的初值。记录定义可分成多列，同样地，每列的结尾需有一个逗号（,）作为结束。

如果有运算式 (expression)，运算式可为区段说明初值。如果初值是大于区段的宽度，编译程序会产生错误信息。

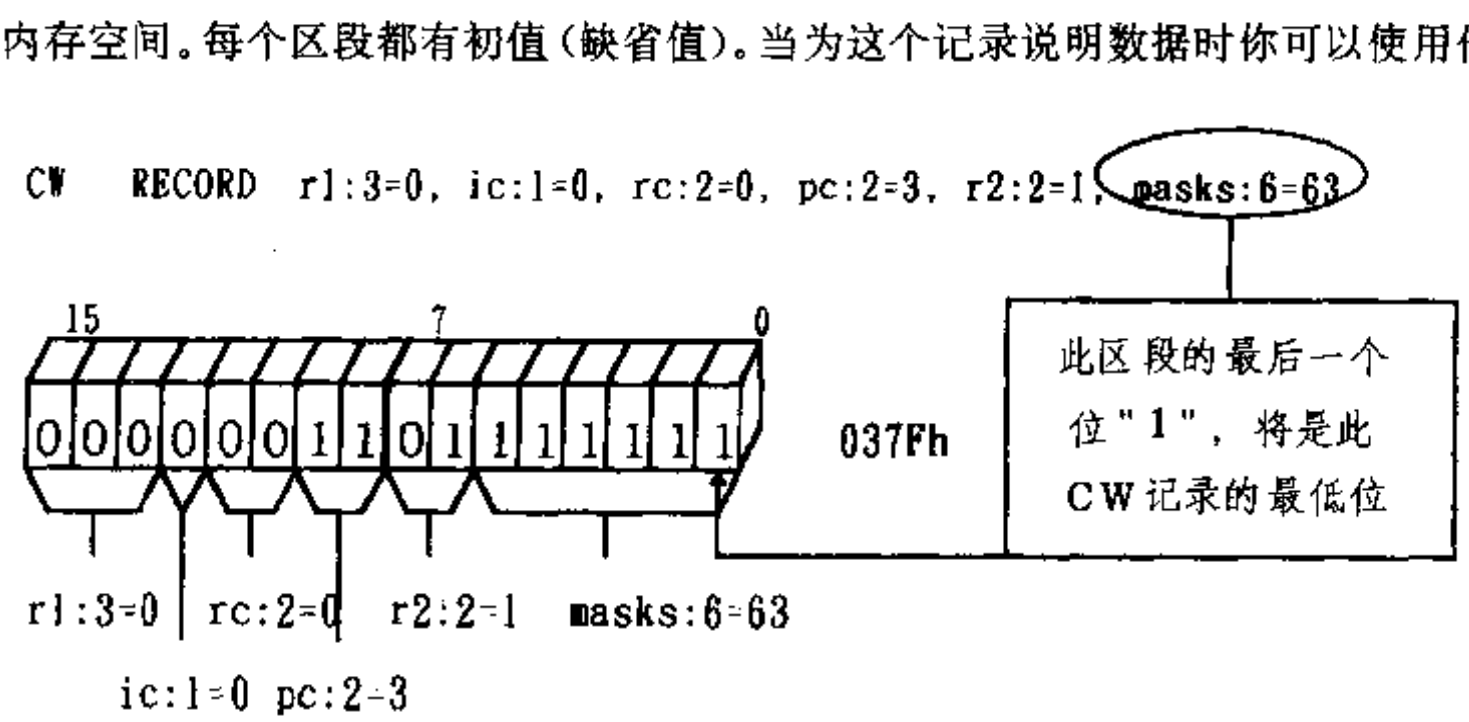
说明中的第一个区段总是此记录最有意义的位。随后的区段是放置在继续之后的位右边。如果区段的总和不是 8、16、32 位，整个的记录会往右移，这样最后一个区段的最后一位将是此记录的最低位。

下面这个实例建立了一个 1 byte 大小的 COLOR 记录类型，共有四个区段-blink、back、intense 和 fore。这个记录类型的内容是在实例之后说明。因为没有初值，所有的位都设为 0。注意它只是一个由编译程序供给的样板，不会在数据段中配置空间。

COLOR　　RECORD　　blink：1，　　back：3，　　intense：1　　fore：3



下面的实例建立了一个有 6 个区段、类型为 CW 的记录。每个说明此类型的记录占据 16 bits 的内存空间。每个区段都有初值（缺省值）。当为这个记录说明数据时你可以使用他们。此



实例之后的位图形说明了记录类型的内容。(3+1+2+2+2=10<16，所以整个记录往右移。前 6 个以零填充)。

5. 定义记录变量

一旦你已说明了一个记录类型，你可以定义此类型的记录变量。对于每一个变量，编译程序会按照类型说明时的格式配置内存。语法是：

[name]　　recordname　　〈[初值[，初值]...]〉

[name]　　recordname　　〈{[初值[，初值]...]}〉

[name]　　recordname　　常数 DUP（[初值[，初值]...]）

recordname 是先前使用 RECORD 伪指令说明的记录类型名称。

对每个区段的 fieldlist 可以是一连串整数、字符常数或运算式(是一个与栏位大小兼容的值)。当你没有指定初值时，你还是要包含"〈　〉"或"〈　〉"。

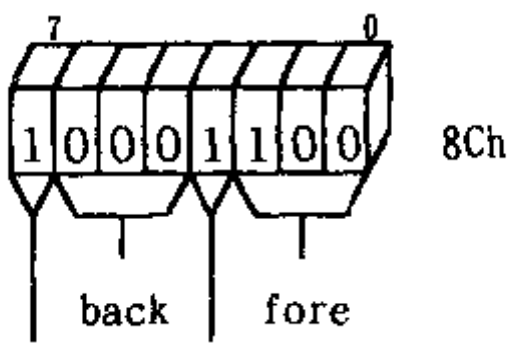
如果你使用 DUP 运算符去初始化许多记录变量，"〈　〉"和任何初值需要以小括号"(　)"括起来。

xmas　COLOR　50　DUP　(〈1，2，0，4〉)

你不需要去定义记录中每个区段的初值。如果初值是空白的，编译程序会自动储存区段的缺省值。如果没有缺省值，编译程序会清除区段中的每一位。

下列实例的定义建立了一个名为 warning 的变量，它的类型是 COLOR 记录类型。在记录中区段的初值设为记录定义时的值。定义的初值可以推翻任何缺省记录值。

```
COLOR     RECORD     blink: 1, back: 3, intense: 1, fore: 3     ; Record
                                                                ; declaration
warning COLOR <1, 0, 1, 4>                                      ; Record
                                                                ; definition
```



6. LENGTHOF，SIZEOF，and TYPE

SIZEOF 和 TYPE 运算符应用在记录名称可返回被记录使用的 byte 数。SIZEOF 返回一个记录变量所占用的 byte 数。在记录说明时你不可以使用 LENGTHOF，但在定义记录变量时可以使用。LENGTHOF 可返回在记录数组中的记录数目，若为单一记录变量则为 1。

```
; Record definition
; 9 bits stored in 2 bytes
RGBCOLOR        RECORD        red: 3,        green: 3, blue: 3

        mov   ax,   RGBCOLOR                  ; Equivalent to "mov ax, 0000h"
;       mov   ax,   LENGTHOF      RGBCOLOR    ; Illegal since LENGTHOF can
```

```
                                                    ; apply only to date label
        mov   ax,   SIZEOF          RGBCOLOR       ; Equivalent to "mov ax, 2"
        mov   ax,   TYPE            RGBCOLOR       ; Equivalent to "mov ax, 2"

; Reclord instance
; 8 bits stored in 1 byte                                    表运行时才初始化
RGBCOLOR2       RECORD red: 3, green: 3, blue: 2
rgb             RGBCOLOR2 (1, 1, 1)      ; Initialize to 00100101ⓨ

        mov   ax,   RGBCOLOR2                      ; Equivalent to
                                                   ;     "mov ax, 00FFh"
        mov   ax,   LENGTHOF        rgb            ; Equivalent to "mov ax, 1"
        mov   ax,   SIZEOF          rgb            ; Equivalent to "mov ax, 1"
        mov   ax,   TYPE            rgb            ; Equivalent to "mov ax, 1"
```

7. 记录运算符

WIDTH 运算符（只能使用在记录）会返回记录或记录区段的位数。MASK 运算符会返回被指定的记录区段所占据的位位置的位标签。也就是会将和位区段相对应的位全设为 1。下面的实例说明如何使用 MASK 和 WIDTH。

```
        .DATA
COLOR           RECORD      blink: 1, back: 3, intense: 1, fore: 3
message         COLOR       (1, 5, 1, 1)
wblink  EQU     WIDTH       blink              ; "wblink" =1
wback   EQU     WIDTH       back               ; "wback" =3
wintens EQU     WIDTH       intense            ; "wintense" =1
wfore   EQU     WIDTH       fore               ; "wfore" =3
wcolor  EQU     WIDTH       COLOR              ; "wcolor" =8
        .CODE
        ...
        mov     ah, message                    ; Load initial          1101 1001
        and     ah, NOT MASK back              ; Turn off          AND 1000 1111
                                               ; "back"                ----
                                               ;                       1000 1001
        or      ah, MASK blink                 ; Turn on            OR 1000 0000
                                               ; "blink"               ----
                                               ;                       1000 1001
        xor     ah, MASK intense               ; Toggle            XOR 0000 1000
                                               ; "Intense"             ----
                                               ;                       1000 0001
        IF      (WIDTH COLOR) GT 8             ; If color is 16 bit, load
            mov ax, message                    ; into 16-bit register
        ELSE                                   ; else
            mov al, message                    ; load into low 8-bit register
            xor ah, ah                         ; and clear high 8-bits
        ENDIF
```

下面的实例继续列出使用记录区段做为操作数或运算式的许多方法：

```
; Rotate "back" of "message" without changing other values

mov   al, message              ; Load value from memory
mov   ah, al                   ; Save a copy for work          1101 1001=ah/al
and   al, NOT MASK back        ; Mask out old bits        AND  1000 1111=mask
                               ;    to save old message        ------
                                                               1000 1001=al
mov   cl, back                 ; Load bit position
shr   ah, cl                   ; Shift to right                0000 1101=ah
inc   ah                       ; Increment                     0000 1110=ah
shl   ah, cl                   ; Shift left again              1110 0000=ah
and   ah, MASK back            ; Mask off extra bits      AND  0111 0000=mask
                               ;    to get new message         ------
                               ;                               0110 0000 ah
or    ah, al                   ; Combine old and new      OR   1000 1001 al
                               ;                               ------
mov   message, ah              ; Write back to memory          1110 1001 ah
```

记录变量通常是被使用逻辑运算符去执行在记录的位区段逻辑运算，如在上例中使用 MASK 运算符。

5.3 寻址未命名的项目

你不需要为每个你定义的数据项目给定一个名称。你可以由先前所定义的变量名称相关的地址存取任何位置。例如，考虑下列的数据定义：

```
TABLE  BYTE  1, 2, 3, 4, 5
       BYTE  ?, ?
       BYTE  8, 9, 10
```

这个语句定义一个表格，包含 10 bytes。前 5 bytes 和最后 3 bytes 都有指定的初值。Bytes6 和 7 被定义成变量。当你的数据项目有常数和变量时，这类的定义是很有用的。你可以用 TABLE+5 和 TABLE+6 去存取 bytes 6 和 7。剩下的 bytes 可用 TABLE+7，TABLE+8，TABLE+9 存取。你通常是使用 TABLE [SI] 去存取表格中不同 BYTES。你必须要确定 SI 的值不超过 9。如果超过，则程序将会存取到不正确的 bytes。例如 SI 为 100，则 TABLE [SI] 将存取到 TABLE 之后 100 BYTES 之处的值。此位置可能是在另一数据段、额外数据段甚至在代码段。

5.4 属性

属性是描述一数据项目的特性。每个数据项目有 5 个不同的属性：

TYPE，LENGTH，SIZE，SEG，OFFSET

除了数据项目外，标记也有 3 个属性（第 9 章）。

TYPE 是每个数据项目所需的 bytes。例如数据项目是 byte，则 TYPE＝1。如果是 words，TYPE＝2。如果是 quadwords，TYPE＝8。

LENGTH 属性是组成数据项目的所有数目。例如有一个变量包含 200 words，LENGTH＝200。若常数包含 50 bytes，LENGTH＝50。

SIZE 属性是配置数据项目所需的总 bytes 数。例如变量包含 200 words，size＝400 (bytes)。常数包含 50 bytes，SIZE＝50 (bytes)。

我们可以得知　SIZE＝TYPE * LENGTH

SEG 属性是数据项目所在的段开头地址。因为数据项目通常是在数据段 (data segment) 或额外段 (extra segment)，所以 SEG 属性通常有相同的值，不是 DS 就是 ES。OFFSET 属性是数据项目第一个 BYTE 的地址 (也就是偏移)。

SEG 与 OFFSET 属性描述了一数据项目的完整地址。当你要使用数据项目的地址时，这些属性比它的内容更有用。

下面有个数据定义实例：

```
TEMP   WORD   200   DUP    (?)
ZEROS  BYTE   50   DUP    (0)
VALUE  QWORD  ?
```

第一个实例定义了一个变量 TEMP，包含 200 words。

TYPE＝2，LENGTH＝200，SIZE＝400。

第二个实例定义了一个常数 ZEROS，包含 50 bytes，初值都是 0。

TYPE＝1，LENGTH＝50，SIZE＝50。

第三个实例定义了一个变量 VALUE，包含一个 quadwords (8 bytes)。

TYPE＝8，LENGTH＝1，SIZE＝8。

5.5 运算符

运算符是使用在编译时去影响操作数的符号。运算符只可被编译程序认识，而和机器指令没有相关。编译程序提供了许多运算符型式。我们只描述应用在数据项目中最有用的几个。

TYPE　LENGTH　SIZE　SEG　OFFSET

为了在操作数中使用这些运算符，可将运算符放置在数据名称之前。编译程序将计算此运算式并将结果替代到被产生的机器指令中。下面有个实例，LIST 定义如下：

```
LIST   WORD   100   DUP    (?)
```

考虑这个指令　　MOV　AX，TYPE LIST

编译程序将会为运算式“TYPE　LIST”产生一个替代值——2。因此产生的机器指令将和下列的指令有同样的效果。

```
MOV   AX，2
```

其余的也是一样。

MOV　AX，LENGTH LIST→MOV　AX，100

MOV　AX，SIZE　LIST→MOV　AX，200

记住，段地址和偏移是 16 位，因此你可不要使用 8 位的寄存器或变量。例如下列的指令

将造成错误：

MOV DH，OFFSET LIST；请使用 16 位的寄存器，如 DX。

属性运算符可使你的程序较易阅读，例如指令：

MOV AX， LENGTH LIST

比（MOV AH，100）能较清楚地表达它的含意。

当要修改程序时，此运算符也是较有用的。例如当你要将 LIST 定义的长度由 100 变到 200，你将不必去改变任何存取到"LENGTH LIST"的叙述，只需要改变任何存取到整数常量（真正值）100 的语句。

算术运算符：+、-、*、/和 **MOD**

你可以使用许多运算符在操作数运算式中去执行简单的算术。实例如下：

```
MOV AX，TABLE+10
MOV AX，TABLE+2*（10+6）
TABLE BYTE 2*1024 DUP（?）
COUNT BYTE 97 MOD （10/8）
```

如果你需要较复杂的复述运算式，你可以使用"（）"去指定哪一个运算将被先执行。你也可以使用"-"符号去表示一个负数。

除法运算符"/"，又称为整数除法（integer division）。它的意思是任何余数都会被忽略。MOD 运算符刚好相反，它是取得整除法的余数。例如"97/10"是 9，"97 mod 10"是 7。

有一点需要注意的是，算术运算符只能使用在编译时所能执行的指定计算。这些运算符不能使用在程序执行时。

算数运算符可使你的程序较易读。例如你要定义一个变量叫 BUFFER，包含 12KB。12KB=12，288（12*1024）。然而如果你使用

```
BUFFER BYTE 12288 DUP （?）
```

数字的意义是不清楚的。如果你使用

```
BUFFER BYTE 12*1024 DUP （?）
```

你的定义将很明显。

也可以将算术运算符和属性运算符混合使用。实例如下：

```
LIST BYTE 512 DUP （?）
BIGLIST BYTE 2*（SIZEOF LIST） DUP（?）
```

5.6 LABEL 伪指令

LABEL 伪指令允许你使用指定的属性去定义一个名称。格式如下：

```
name LABEL type
```

此名称不会在机器语言程序中占据任何空间。它可提供你以另一个方式存取在程序中的一个特别的位置。以这样方式定义的名称被称为 LABEL。实例如下：

```
BNAME LABEL BYTE
WNAME WORD 100 DUP （?）
```

第一条语句定义了一个 LABEL；BNAME（它指明到包含 bytes 的数据项目）。第二条语

句定义了一个数据项目 WNAME（包含 words）。

因为 LABEL 伪指令不会占据任何机器语言程序空间，所以 BNAME 和 WNAME 有相同的偏移。因此，你可以使用 BNAME 去存取 BYTE 的数据项目和 WNAME 去存取 WORD 的数据项目，这就好像一个数据项目有两个名称。例如要搬移 WNAME 的第二个 WORD 的内容到 AX 寄存器：

```
MOV  AX, WNAME+2
```

（记住每个 WORD 是 2 BYTES。）然而如果你要搬移此 WORD 的第一个半部（1 BYTE）到 AH 寄存器，你不可以使用

```
MOV  AH, WNAME+2
```

因为编译程序不允许在包含 WORD 的数据项目内存取 BYTE 的数据。而你可以使用 BNAME：

```
MOV  AH, BNAME+2
```

这和使用 PTR 运算符去强定这个类型是相同的：

```
MOV  AH, BYTE PTR WNAME+2
```

不要忘记，处理器和编译程序在一个 WORD 内存储 BYTES 是以反转的顺序存储。

当你定义了一个标记，你可以使用任何一种类型——BYTE、WORD、DWORD、QWORD 或 TBYTE。

5.7 EQU 伪指令

EQU 伪指令允许你定义一个符号，在程序编译时代表一个指定的值。

```
name  EQU expression
```

例如下列的伪指令告诉编译程序去考虑符号 K 代表 1024：

```
K     EQU     1024
```

每个 EQU 出现的地方被考虑去代表“1024”。我们可以说这样一个语句如同相等(equate)。

当你的程序需重复使用相同的值时，EQU 是很有用的。你可以使用 EQU 去指派一个值给一个符号，然后可在程序任何地方使用此符号。例如说你使用 1K（1024）去定义许多数据项目：

```
ITEM1  BYTE      1024  DUP  (?)
ITEM2  BYTE  2*1024  DUP  (?)
ITEM3  BYTE  3*1024  DUP  (?)
```

如果你先前已定义了符号 K，你可以使用

```
ITEM1  BYTE      K  DUP  (?)
ITEM2  BYTE    2*K  DUP  (?)
ITEM3  BYTE    3*K  DUP  (?)
```

了解 EQU 并不会在机器语言程序产生任何东西是很重要的。他们只是编译程序的指令（编译程序提供此缩写是为了在编译过程中使用）。

当你多条语句需依照某个基本的值时，EQU 特别有用。使用 EQU 你可以借助符号名称去代表这个值。如果这些值需要改变，你所要做的只是改变 EQU 并再编译此程序。

例如下列的 EQU 实例，设置了 LSIZE 为 100：

```
LSIZE     EQU     100
```

在程序中可能有许多语句可利用这个值，例如：

```
WNAME     BYTE     LSIZE     DUP     (?)
WLIST     WORD     LSIZE     DUP     (?)
          MOV      AX,    LSIZE
          MOV      BX,    LSIZE
```

等等。然而使用 EQU 最大的用处，就是让程序通过使用有意义的名称去替代数字，使程序较易读。

使用 EQU 规则

如前面所解释的，EQU 使用的格式如下：

```
name   EQU   expression
```

此命名是遵循标准的规则（见 5.9）。要确定名称是独一无二的，且不能是保留字。expression 可以是 0 到 65535 之间的无符号数。你也可以使用算术和属性运算符，你也可以以十六、十、或二进制指定数字。二进制对指定位区段是特别有用的。

使用实例如下：

```
MAX_VAL   EQU   157
K         EQU   1024
TEN_K     EQU   10 * K
ITS       EQU   1000011110100110B
HEXVAL    EQU   0A3Fh
LTYPE     EQU   TYPE LIST
```

你必须确定编译程序了解你所使用的任何名称。因此在使用"10 * K"的实例中，名称 K 必须被定义在别的地方。相同地，在上个例子中，LIST 必定是一个数据项目的名称（变量名称），否则将产生错误。

许多编译程序允许你指定非数值的值，如一连串的字符。例如你可以使用下列语句：

```
MESSAGE    EQU    "Hello, how are you?"
OUTPUT     BYTE   MESSAGE
```

因为 EQU 是对编译程序下的指令，不会产生机器指令，你可以放置在任何地方。最好的主意是搜集你所有的 EQU 在靠近程序开头的一个分离的区段。这可使得程序较易阅读。然而也有许多 EQU 是出现在数据段。这将在下一个章节详细说明。

5.8 地址计数器：$ 和 ORG 伪指令

编译程序保持一个称为地址计数器 (location counter) 的值，它包含下一个可用位置的偏移。当处理器开始处理程序时，此位置计算器被设为 0。每次编译程序为数据或指令安置一个内存区段时，此地址计数器值会增加。

下面有个实例，编译程序开始处理程序时，地址计数器开始在 0H。编译程序遇到伪指令

BYTE 32 DUP ("STACK—")

会设置 100H（256＝32＊8）bytes 的存储区段。地址计数器现在将有 100H 的值。

在程序中，你可以通过使用 $ 去存取目前地址计数器的值。例如你定义

CURRENT EQU $

编译程序将定义 CURRENT 成为目前 EQU 被处理时的地址计数器的值。字符"$"通常被用来提供定义一连串字符长度一个容易的方法。如下面的语句：

MSG BYTE "Hello"

L_MSG EQU $-MSG

$-MSG 运算式表示目前地址计数器的值减去 MSG 偏移，换句话说就是 MSG 的长度。此技巧是非常方便的，因为它允许你可以只改变字符串，而不用改变 EQU。这也使得不需要去计算字符数。例如，你定义 L-MSG

L-MSG EQU 5

如果你改变 MSG，你还必须再计算它的长度。

你也可以通过使用 ORG（origin）伪指令设置地址计数器到一个真正的值。格式如下：

ORG value

此伪指令常用在指定编译程序开始在大于 0 的偏移。下列的伪指令是使用在程序的开头：

ORG 100H

它告诉编译程序在偏移 100H 开始编译。此伪指令会在程序的开头预留 100H（256 bytes）bytes 的空间。下面的实例列出了如何使用 EQU 与放置的位置。

```
.DOSSEG
.286
.MODEL small
.STACK 1024
; ——— Equate———
CR              EQU     0DH             ; ASCII code for "carriage return"
LF              EQU     0AH             ; ASCII code for "line feed"
DISPLAY         EQU     0001H           ; file handle for the display
                                        ; 01H 表示输出至屏幕
.DATA
                MSG     BYTE    " This is a message.", CR, LF
                L_MSG   EQU     $-MSG
.CODE
                main    proc
                        .STARTUP
                        ;
                        mov bx, DISPLAY     ; 输出的文件代码，此例 01h 表屏幕
                        mov cx, L_MSG       ; 输出的字符数
                        lea dx, MSG         ; 使用 mov dx, offset MSG 也可以
                                            ; DS：DX 为 MSG 的偏移
                        mov ah, 40H         ; 第 40H 号功能
```

```
                int 21H                 ; Call DOS
                ;
                .EXIT
        main    endp
END
```

5.9 变量命名规则

变量名称是你在程序中所选择的表示不同地址的符号。你必须遵照下列的规则为你的变量或符号命名。

●变量名称可以使用英文字母（'A'～'Z','a'～'z'）、阿拉伯数字（'0'～'9'）以及下列的特别字符：

? @ _ $

●变量名称的第一个字符不可以是数字('0'～'9')。因为这可使得编译程序去辨别是变量还是数字（数字的第一个字符总是数字）。

●不要使用‘@’字符做为变量的第一个字符。因为由‘@’字符开头的变量通常都有不同的目的，如@data。

●你可以按照你所喜欢的方式去命名，但编译程序只认得前 31 个字符，也就是说，如果你有两个变量名称的前 31 个字符是相同，编译程序将视为同一个。这点要特别注意。

下面是合法的实例：

HELLO $MARKET A1234 MAN_NAME PART_3

下面是不合法的实例：

MAN-NAME 3_PART

第一个是因为使用‘-’字符，第二个是因为以数字“3”开头。这是常犯的错误，要注意。

你也不可以使用保留字去作为变量的名称。因为保留字可能是指令或伪指令的 opcode，或寄存器的名称。例如，你不可以使用 MOV 去作为变量名称，因为它是指令的 opcode。你也不可以使用 AX，因为它是寄存器名称。你可以查阅附录 E。

5.10 指定数字规则

除了标点符号，操作数还包含符号（变量或寄存器名称）与数字。编译程序允许你指定不同种类的数字。你可以使用十（decimal）、十六（hex）、二（binary）进制数字。

十进制数字包含（'0'～'9'），如 mov ax，123。结尾不需加‘D’。

十六进制数字包含('0'～'9'及'A'～'Z')，如 mov ax，123H。如果第一个字符为'A'～'F'，则开头需加‘0’。这是告诉编译程序这是一个数字，不是一个变量名称。

二进制数字包含（'0'～'1'），如 mov al，00001111B。结尾需加上‘B’或‘b’以表示是一个二进制数字。

第 6 章　基本字符输出、输入

前面章节介绍的指令在执行过程中的变化，均需使用 debug. EXE 去观察。本章将介绍基本字符、字符串输出、输入方法，以后就可舍弃功能不强的 debug，改在屏幕作输出了。

INT 指令在汇编语言中的运用，是相当频繁且重要的，而字符输出、输入是做好程序与 User 之间界面的重要工具，学好本章将使您的汇编语言程序大大改观，放心，您将会非常轻松、容易地学好本章，现在就让我们带你进入这奇妙的 INT 世界。

PC 若需要输出、输入服务时，会调用操作系统的服务程序。这些服务程序通常是由 DOS 与 ROM BIOS 所提供。PC 内要载入 DOS 才能提供 DOS 服务程序，而 ROM BIOS 在 IBM PC 里本身就有提供。完整中断说明在附录中，你若兴趣，请欢迎查阅。

6.1　中断

处理器是被设计成使用两套系统去处理所发生的不可预期事件。无论何时，计算机都会辨认一个需要立即处理发生的事件所传回来的信号。这样的信号被称为中断（interrupt）。

计算机必须处理许多不可预期发生的事，大部分是来自硬件。

●键盘被按时传回的信号。

●打印机没纸时传回的信号。

●磁盘传递数据完成时传回的信号。

●内存晶片有错误发生时传回的信号。

经常发生在软件的不可预测事件：

●程序执行时尝试去除零。

当中断发生时，处理器必须执行一个特别的子程序，称为中断服务程序（interrupt handler）。

每个中断都有它自己的服务程序，处理器会暂时把目前执行的程序停下来，去处理与此相对应的服务程序。一旦服务程序执行完成，处理器会继续执行刚才未完成的程序。由于现在的处理器执行的速度相当快，所以虽然经常有中断发生，但其执行所占用的时间都相当的短，我们感觉就好像没有发生中断一样。

中断发生的原因有很多种，通常是外围设备，如：键盘、磁盘驱动器、打印机等。Intel CPU 可以识别两种型式的中断。

①硬件中断：通常是外围设备向 CPU 要求中断服务。

②软件中断：通常是程序向操作系统要求 Input、output 服务时。所以一个程序执行的过程中常会有中断情况发生。通常占用时间极短，并不影响程序执行，各位大可放心。

6.2 软硬件中断

源自硬件装置发出信号的中断称为硬件中断（hardware interrupt）或外部中断（external interrupt）。而源自执行中的程序所发生的中断称为软件中断（software interrupt）或内部中断（internal interrupt）。

服务硬件装置是一件相当复杂的工作，最好是留给内建在DOS的服务程序或相关硬件去处理较好。然而，作为汇编语言程序员，只需要使用许多软件中断即可。这将允许你只需依赖操作系统的资源去执行重要但技术上较困难的工作。

严格地说，软件中断并不是一个中断，只是软件中断执行的行为（操作）很像硬件中断的执行而已。计算机里有软件中断服务的提供，有两个优点：

（1）程序的编写会变得较容易。

（2）提供可靠性。

我们不必考虑硬件的兼容问题，只须调用软件中断服务。因为软件中断服务的方式均大致兼容，各种硬件如何完成服务的要求，就不关我们的事了。

6.3 INT 指令

中断有两个目的：

1. 允许计算机对不可预期发生的事件做出回应。大部分来说，这些事件是由许多不同的硬件所传回来的信号所造成的。因此中断可以说是允许处理器去提供此硬件服务的方法。然而中断也可以被使用在完全不同的目的的情况下。

2. 我们可以通过一个特别的号码去作为此中断的信号，那么程序将不需要知道服务程序的名字及位置，而能够调用服务程序。

为什么程序需要去调用这样一个服务程序？因为DOS与BIOS为程序提供了许多服务。为了要利用这些服务程序，程序必须通过提供这个适当的中断信号得到服务。事实上有一个指令INT可以做到这一点。语法如下：

INT number

此number数字范围为0～0FFH，此INT指令会call一个中断服务程序。一般在执行INT指令前，会在AH中先放一个数字，稍后将对此作出解释。通常我们向DOS或BIOS要求如IO（input-output）、文件和屏幕等服务时，都会用到INT指令。

6.4 中断向量表（Interrupt Vector Table）

在内存（RAM）中最低地址的1024 bytes为中断向量表。实际上，中断向量表是一群地址的汇编，每一组地址是由4 bytes构成的。

segment：offset ； 2 bytes：2 bytes

1024/4＝256＝0FFh，可知最多有256组地址。而INT number的number就有0～0FFh大小的范围。每一组地址是每一个相对应中断服务程序的开关地址，且每一个中断服务程序

还可能细分有许多子服务程序，也是由数值来指定使用哪一个子服务程序。只要将数值放在AH就可以。当然也可能会传一些参数给服务程序，传参数的方法只要利用其余的寄存器即可解决。

6.5 BIOS

BIOS (Basic Input/Output System)，事实上是存储在ROM中一系列的复杂程序。它主要提供三个重要功能。

(1) BIOS包含每次开机时会自动执行的自我测试 (Power-On Self Test; POST)。POST会检查计算机的部分元件，包含内存。主要是要在开始使用机器之前检查硬件有无错误。

(2) BIOS包含许多特别的程序，称为设备驱动程序 (device drivers)。它提供对于许多不同的硬件设备的标准界面。如果没有设备驱动程序，你可能只使用一个硬件就必须做大量复杂的程序设计。一般来说，设备驱动程序可避免程序员去了解不同硬件设备的特性。除了这些内置的设备驱动程序之外，DOS也允许你使用可安装的设置驱动程序，它可在每次开机时设置。它允许在硬件设计有较进步的改进。可安装的设置驱动程序可以通过在CONFIG.SYS文件中使用DEVICE命令指定。

(3) 设备驱动程序提供一组有用的服务程序集合。这些服务程序是软件中断所需要的。最重要的是，在你自己的程序中可以要求这些服务 (通过使用INT指令与适当的号码)。然而大部分的服务相当的低级，你应该不会直接去使用他们。他们实际上较多地提供给操作系统所使用。

6.6 DOS

许多人认为操作系统是作为计算机执行的主要控制程序。这是正确的，但作为一个程序员，你也可以考虑操作系统所提供的服务。为了需要这些服务 (DOS)，你可以使用INT指令去提供软件中断的信号，可以将DOS想成是你的程序与硬件的界面。DOS是被设计去执行在你的程序中所需要的重要、复杂的工作。因此无论何时你都应该记住尽量让DOS做，你不需要知道它真正是如何执行工作的。例如，你可能需要DOS去读部分的磁盘文件，但不需要担心这需要去执行几百个低级的指令。当需要时，DOS会调用BIOS帮忙，它是完全自动地完成的。

6.7 DOS Function Call

INT (20H～3FH) 是保留给DOS的服务程序使用。其中最常用与最重要的是INT 21H。在INT 21H中有130个不同的function提供给程序服务 (指在DOS 6.2)，我们称为Dos Function Call。因为所有的功能调用是通过相同的中断执行，因此可规定一个标准的方式去提供程序的服务。每个功能调用都有一个调用号码表示，只要将此调用号码放在AH寄存器中，即可使用此功能调用。例如，功能调用19H可决定目前缺省的驱动器。下面的指令可执行此功能调用：

```
; determine current default drive
MOV  AH, 19H
INT  21H
```

事实上，IBM 和 Microsoft 都鼓励程序员只使用 Function call 去要求服务。(这就是为什么正式的参考手册只描述了少数其它的 DOS 中断，且没有描述 BIOS 中断)。

大部分的功能调用都需要输入与产生输出。然而，功能调用并没有存取调用程序的数据区段。因此，所有的信息都是通过寄存器传递。

在你使用功能调用之前，仔细查阅一下，去看看寄存器应如何正确地设置，执行完后应该传回那些信息。你需要 DOS 参考手册，或一本讨论每个中断细节的程序设计书籍。

对于输入，功能调用总是需要在 AH 存放调用号码。一般的规则是，DL 通常是占有一个输入值，DX 通常是被使用去存放在数据段字符串的偏移。对于输出，通常是通过 AX、AL、DL 寄存器传回。当然也有一些例外。

大部分的功能调用都有一个特征就是当有错误发生时，他们会向调用程序发出信号。如果有错误发生，进位标志 (Carry Flag；CF) 会被设为 1。如果没有，CF 将清为 0。因此我们可以简单地在 INT 指令之后使用条件转移 (JC 或 JNC) 去测试是否有错误。

使用中断 INT 21H 去要求 DOS 的功能调用和调用一个正常的子程序是一样的。

功能调用不会改变寄存器或标志值，除了需通过寄存器传回参数。(有一个例外：你不可以假设功能调用会保护 AX 的值。)

一旦 DOS 得到控制权，它会使用自己内部的堆栈。然而中断使用调用程序的堆栈足够大，应该要使堆栈至少比程序本身所需大 200H (512) bytes。我们先介绍有关字符输出、输入的 0～0CH 的 function call。

6.7.1 01H：由键盘输入一字符且显示在屏幕上

AH＝01H，等待由 console (控制台) 输入一字符，然后存储在 AL 中。此功能会检查 CTRL-BREAK (CTRL-C) 且输入的字符会 echo (回应)，即显示在屏幕上，且读入的值会存储在 AL。

```
mov     AH,   1      ; 等待由键盘输入一字符
INT     21H,         ; 执行中断服务程序
mov     char, AL     ; 在 char 变量中存储字符
```

所谓 console 即指键盘，上例假设数据段有一个 char 变量说明；1 bytes。DOS 会将用户输入的字符，存储在一环状 buffer，即一块可以存储 15 个字符的内存空间。如果输入太多，DOS 处理不了时，计算机就会叫了，告诉你不要再输入了。多余的输入将忽略。

6.7.2 02H：输出字符至屏幕

AH＝02H，把要输出的字符放在 DL 中，会检查 CTRL-BREAK。

```
mov     DL,   'A'    ; 准备输出的字符'A'
mov     AH,   2      ; 输出一字符至屏幕
int     21H          ; 执行中断服务程序
```

6.7.3 05H：打印机输出

AH＝05H，把将要输出至打印机的字符放在 DL。同样会检查 CTRL-BREAK，缺省时，打印机是接在 LPT1。现在打印机都有内部 buffer (即打印机有它自己的内存)，打印机会等

到 buffer 满了或接收到一归位字符 (carriage return；0DH)，才会作打印的操作。所以可以送一个 0DH 去强迫打印。

```
mov     AH,    5      ；执行打印机输出   ①
mov     DL,    'A'    ；输出字符'A'      ②
int     21H           ；执行中断服务程序③
mov     DL,    0DH    ；输出归位字符     ④
int     21H           ；执行中断服务程序⑤
```

由于第①列已输入 AH＝5，所以第⑤列之前就不需要再写一次（mov AH，5）第⑤列直接写 int 21H 就可以省去一列。

6.7.4 06H：控制台直接输出、输入

AH＝06H，有两种模式。①输出模式时，DL 需设为 0FFh，且 DOS 并不会等待输入。输入字符会被放至 AL，若用户没有输入或来不及输入，零值标志（Zero Flag；ZF）将设为 1。②输出模式时，DL 不能为 0FFh，输出的字符放在 DL，输出至屏幕。两种模式都不检查 CTRL-BREAK。

①输入模式

```
mov     AH,    6
mov     DL,    0FFH     ；执行输入模式，且并不等待
int     21H             ；如果有按键，将存放在 AL，若没有则
                        ；ZF＝1
```

②输出模式

```
mov     AH,    6
mov     DL,    '*'      ；输出字符'*'至屏幕
int     21H
```

ASCII 字符 0～20H 为控制字符，若使用此功能，将被解读为控制字符。如 AL＝0DH，则光标将归位（移至列开头），若 AL＝0AH，光标将移至下列同位置。若同时输出 0DH 与 0AH 字符，则光标将移至下列开头，如同在某些编辑器里按 Enter 换列情况相同。

6.7.5 07H：控制台直接输入且没有回送

AH＝07H，等待由键盘输入一个字符，此字符并不显示在屏幕上，且不检查 CTRL-BREAK，输入的字符存储在 AL 中。

```
mov     AH,    7        ；等待键盘输入，屏幕上看不到输入字符
mov     21H             ；并不检查 CTRL-BREAK
```

若程序为输入密码并不想为其它人看见时，常应用此服务程序。像读特殊键，如方向键时，也可使用。

6.7.6 08H：控制台直接输入且没有回送

AH＝08H，等待控制台输入一字符，也不显示在屏幕上，但会检查 CTRL-BREAK，输入之字符同样存储在 AL。

```
mov     AH,    8        ；会等待，不回应
int     21H             ；检查 CTRL-BREAK
```

6.7.7 09H：字符串输出

AH=09H，将以 DS：DX 开头地址的字符串输出至屏幕，一直输出至'$'字符才结束输出。注意，'$'并不会被输出，且 DOS 会识别此控制字符。

```
.DATA
        string BYTE    'Example of string output', 0DH, 0AH,'$'
.CODE
...
mov     AH,   9
mov     DX,   OFFSET string   ; 将字符串的偏移放至 DX 中
int     21H
```

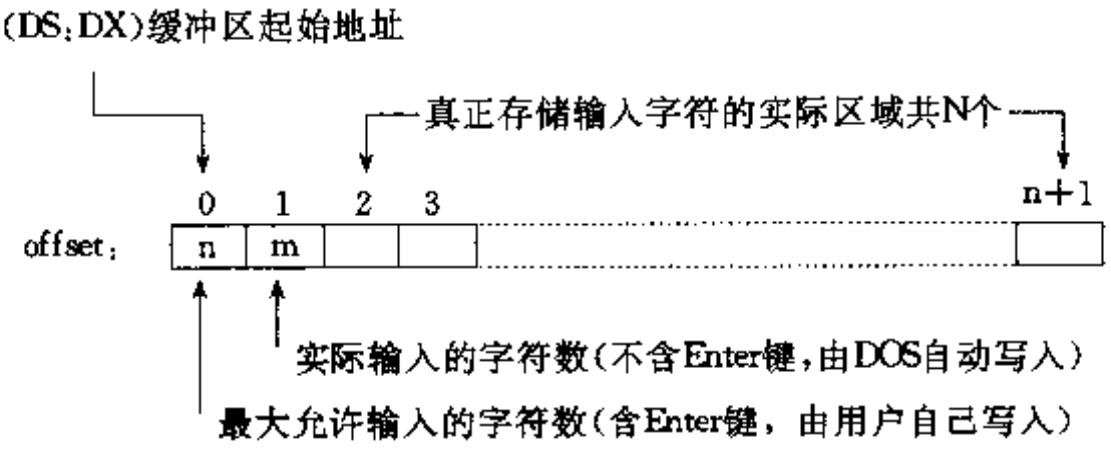
此功能的缺点是须在字符串结束处自己设'$'字符，也就是无法输出'$'字符。如果没遇到'$'字符就会一直输出。特别注意要记得，在字符串结尾处加上一个'$'，以终止字符串的输出的操作。

6.7.8 0AH：字符串输入

AH=0AH，等待自键盘输入一字符串，以 Enter 结束字符串的输入，所以最多可输入 254 个字符，含 Enter 算在内则为 255 个字符。会检查 CTRL-BREAK，输入字符串过程中可使用 BACKSPACE 键去修改输入的字符串。

DOS 还会过滤掉一些特殊键如方向键、PgDn 等等，并不会将它存储在 Buffer 中。输入的字符会存储在以 DS：DX 为开头地址的 Buffer（缓冲区）中。

缓冲区（Buffer）结构须注意：




```
max-chars   BYTE   80      ; 用户自订允许输入的最大字符数
nums-input BYTE    ?       ; DOS 会自动写入真正输入的字符数
Buffer      BYTE   80 dup (0); 字符真正存储的区段
...
mov     AH,   0AH
mov     DX,   OFFSET max-chars   ; 将缓冲区起始地址偏移写入 DX
int     21H
```

执行完后，DOS 会在最后一个真正存储字符的下一个地址写入0DH

Enter

```
max-chars nums-input
↓              ↓
05             03    41    42    43    0D……
```

若用户已输入了 n－1 个字符时，计算机会发生“哔”声音提醒你，缓冲区已满，叫你不要再输入了。多余的字符将被舍弃掉。

6.7.9 0BH：键盘缓冲区状态

AH＝0BH，检查是否有字符存放在键盘缓冲区。如果有，DOS 会传回 0FFH 在 AL，若没有，则放置 00H 在 AL。此功能会检查 CTRL-BREAK。

```
mov     AH，0BH      ；检查键盘缓冲区状态
int     21H          ；AL＝0FFH，表示有字符存在
                     ；AL＝00H，表示没有字符存在
```

6.7.10 0CH：清除键盘缓冲区，并等待输入

AH＝0CH，将键盘缓冲区清成空白，并执行指定在 AL 寄存器所表示的输入功能。

若 AL 是（01H、06H、07H、08H、0AH）以外的值，如 00H，则仅做清除操作，而不做输入操作。若做输入操作，则传回值仍存放在 AL 中。

```
mov   AH，0CH
mov   AL，08H      ；调用 DOS INT 21H，第八号输入功能
int   21H
```

如果你希望避免 User 输入过多的命令（即按键过于频繁）而影响执行输入的操作，此功能特别有用。

6.8 扩展码（Extended Code）

在键盘上有许多按键并不是传回 ASCII 码，例如 F1～F12、方向键、PgUp、PgDn、Home、End 等。键盘上每一个按键都有一个不同位置，如键盘左上角的 1 与右边的 1。当一般键被按下时，在 key buffer 中会存放 2 bytes 的数据，第一个 byte 为 ASCII 码，第二个 byte 为扫描码（scan code）。DOS 一般只传回 ASCII 码。若有一些特殊键被按下时，则存放在 key buffer 的两个 bytes，第一个为 0，第二个为扩展码（也可称为 extended code），DOS 会将两个码都传回。程序中若要判别按键是否为特殊键时，须先判别传回值（AL 中），是否为 0。若为 0，则须再读入判别

下一码，以决定是何特殊键被按下。

```
mov   AH，7    ；无回应会等待。
int   21H      ；假设是特殊键。
int   21H      ；再读一次，扩展码在 AL。
```

要读特殊键的场合，function 7、8 特别有用。因为这两项功能并不回应，所以不会破坏屏幕上的格式，而 function 1 会回应至屏幕，所以并不适用。

下列的 CH6-1.ASM 示范由键盘输入字符串。它会先在屏幕上打印一列提示字符串，等待用户输入一个不超过 80 个字的字符串，并以 Enter 结束输入。若超过 80 个字，超过的字将被忽略掉。最后会等待用户按任意键退出。

```
.DOSSEG          ; ************ CH6-1.ASM **************
.MODEL SMALL
.STACK 1024
.DATA
    message1 BYTE 'Please input a string→', 0Ah, 0Dh,' $'
    label1   BYTE 80
    label2   BYTE 0
    label3   BYTE 80 dup (' ')
    message2 BYTE 'String input OK!, Press any key to exit !',' $'
.CODE
    MAIN PROC
        .STARTUP
        mov ah, 9
        mov dx, offset message1      ; 假设地址在 DS: DX
        int 21h
        mov ah, 0Ch                  ; 0Ch 也可以写成 13
        mov al, 0Ah
        mov dx, offset label1
        int 21h
        mov ah, 2
        mov dl, 0Ah
        int 21h
        mov ah, 9
        mov dx, offset message2
        int 21h
        mov ah, 7                    ; 等待输入，但并不将输入字符回应
                                     至屏幕上
        int 21h
        .EXIT
    MAIN ENDP
END
```

在提示字符串 message1 的结尾有 0Ah、0Dh，这将使光标跳至下一列开头，等待用户输入一个字符串。在等待输入之前先使用 0Ch 功能清除键盘缓冲区，然后再执行 AL 中的 0Ah 字符串输入功能。字符串的长度由用户规定在 label1 的地址。DOS 会将真正输入字符串的长度存放在 label2。而字符串真正存储的地址是由 label3 的地址开始存放。

由于字符串输入执行完后（Enter 键按完之后），光标并不会移动还是停留在此列的开头，所以我们先执行 02h 字符输出功能，先输出一个跳列字符（0Ah），迫使光标跳至下列的同一个位置（此例为第一行），再输出字符串 message2，最后再执行 07h 等待输入功能。此功能并不回应输入的字符在屏幕上，所以并不影响屏幕格式。

在代码段中，我们习惯定义一个 main 程序，代表这是一个主程序。在此实例中也可以省略 MAIN PROC 与 MAIN ENDP，照样可以执行，只不过 MAIN 代表程序的进入点。使用

.STARTUP 与 .EXIT 可省略我们要为程序所做的一些繁琐的启始与结束的工作。

如果是 MASM 5.X 老版本的用户，记住不必在 END 伪指令之后加上 MAIN。而 .DOSSEG 与MASM 5.X 的 DOSSEG 具有相同的效果，都可以使用。BYTE 是新式写法（同 DB），同样地两者皆可使用，其余的 WORD，DWORD... 以此类推。

6.9 ASCII 控制字符

在 ASCII 字符集 0～20H 有许多控制字符（如图 6-1 所示），这些控制字符 DOS 会将它视为特殊用途，并不会显示在屏幕上。

通常在许多文本文件中每列的结束都会有 0DH、0AH 字符出现。DOS 在做屏幕输出时，若读到 0DH、0AH 这两个字符时，会将光标跳到下一列开头处继续输出。下面实例将字符串分别输出在不同列：

```
str1   BYTE 'The str1 is on 1th line ',
       0DH, 0AH
str2   BYTE 'The str2 is on 2th line ',
       0DH, 0AH,'$'
        :
       mov AH, 9
       mov DX, offset str1
       int  21H
        :
```

Hexadecimal	意义
08	Backspace
09	tab
0A	Line feed(跳列)
0C	Form feed(跳一页;打印机使用)
0D	Carriage return(归位;Enter key)
1B	Escpae(Esc)

图 6-1 DOS 控制字符

图 6-2 中，四个 DOS 提供的标准字符输入功能，且都是通过 AL 存储输入的字符。至于其它字符输出功能，皆是将要输出的字符放在 DL，然后再将 DL 所指定输出的字符输出至外围设备。

函数号码	等候输入	显示在屏幕	接受 Ctrl-Break
(AH=)01H	Yes	Yes	Yes
06H	No	No	No
07H	Yes	No	No
08H	Yes	No	Yes

图 6-2 DOS 标准字符输入功能

大部分 DOS 提供的服务程序都会检查是否有 CTRL-BREAK 被按下，如果有，则产生 INT 23H 中断，停止程序执行，直接返回 DOS 控制之下。

按 CTRL-BREAK 与 CTRL-C 是同样效果。

注意 06H、07H 功能不会检查 CTRL-BREAK。

6.10 宏（Macro）

宏是一个符号名称。它可以是一连串的字符称为本文宏（text macro），或由一个或多个语句组成的宏过程或宏函数（macro procedure or function）。编译程序会扫描原始程序是否有先前定义的宏名称被当成语句使用在程序中。如果发现这种情况，会将宏的内容替换宏名称，即将宏所代表的内容插入宏名称出现的地方。这样你可以避免在程序中多次编写相同的程序码。

6.10.1 宏过程 (Macro Procedure)

如果你的程序必须执行许多次相同的工作，可以通过编写一个宏过程，避免每次都要重复键入相同的语句。可以将宏过程（一般称为宏）想成会自动重复产生文字的文字处理技巧。

宏类似子程序，它是由一个或多个汇编语言语句所组成，就像是一个语句的集合。先举个例子就能明白它的用法。常要求 User 输入一字符时都要写两行语句：

```
①mov AH, char      ; char 可能为 01H、06H、07H、08H
②int  21H
```

现在我们可以将它写成如下格式：

```
(1) getche MACRO
    mov  AH, 01H
    int   21H
ENDM getche
```

```
(2) getch  macro
    mov  AH, 07H
    int   21H
ENDM getch
```

(1) getche 执行时会将输入的字符回应至屏幕，且会等待用户输入，并接受 CTRL-BREAK。

(2) getch 执行时并不会将输入的字符回应至屏幕，也会等待用户输入，并接受 CTRL-BREAK。

哈哈：各位学过 C 语言的朋友，是不是觉得上面这两个例子很熟悉。只要将上述的宏写一遍，以后在任何场合如果需要输入，只要写一列 getch 或 getche（写哪一个当然要视你需要而定）即可。且你要写几次就写几次，完全看你高兴。除了可读性增高，还可以少写好几行，节省程序设计的时间。所以我迫不及待地要赶快告诉你这个消息。再介绍一个常用的宏例子：

```
putchar  MACRO  char
    mov  AH, 2
    mov  DL, char
    int   21H
ENDM  putchar
.CODE
    ...
    putchar  'A'
    putchar  42H
    putchar character
    mov  AL, 44H
    putchar  AL
    ...
.DATA
    character BYTE 'C'
```

嘿！嘿！怎么这次看起来又像是 C，且 macro 右边多了一个 char 变量。不错，宏确实可以传参数（传值）。如果有两个参数或以上，可用逗号（,）隔开参数，也可以将宏改写成下例：

```
putchar  MACRO
```

```
        mov  AH, 2
        int  21H
        ENDM  putchar
        ...
    .CODE
        mov  DL,  'A'
        putchar
        mov  DL,  42H
        putchar
        mov  DL, character
        putchar
        ...
    .DATA
        character BYTE 'C'
```

6.10.2 建立宏过程

宏一般是写在 .code 之前，但也可写在 .code 区段里。不过要写在调用它之前，不然连接时会发生错误。

你可以定义一个宏过程，将想要的语句放在 MACRO 与 ENDM 伪指令之间即可。如下：

```
macroname  Macro
语句
ENDM  [macroname]
```

●macroname 为宏名称，可自行定义。

●整个宏由 macroname、Macro 和 Endm 构成开头和结尾，语句写在中间。

●ENDM 之后的宏名称可省略不写，嘿！又轻松了不少。

例如，假设你要程序遭遇到错误时“哔”一声，你可以定义一个 beep 宏，如下：

```
beep  MACRO
    mov  ah, 2   ;; Select DOS Print Char function
    mov  dl, 7   ;; Select ASCII 7 (bell)
    int  21h     ;; Call Dos
ENDM
```

你应该看到，上例在宏中的注解使用了两个分号";;"。因为在列表文件中当宏被展开时，只使用一个分号的注解，会在宏被展开的地方，连带也将注解含入。若使用两个分号可避免将注解也一并展开，而使得列表文件较易读。

一旦你已定义了一个宏，你可以使用宏名称当成语句，在程序的任何地方去调用它。当然，你也可以将宏写成一个子程序，然后再用 CALL 指令去调用它。

6.10.3 传参数给宏

你也可以定义参数给宏。完整的宏过程定义语法包含参数列（parameterlist）：

```
name MACRO [parameterlist]
语句
```

```
    ENDM
```

●Parameterlist 可以包含任意数目的参数。

●你不可以使用保留字当做参数名称，除非你使用 OPTION NOKEYWORD。参数是选择性的，可有可无，所以用中括号。记住用逗号隔开。参数名也是自定的。

●当然在调用时，若有两个或两个以上参数，也要用逗号隔开。实例如下：

```
    writechar   MACRO   char
        mov   ah, 2
        mov   dl, char
        int     21h
    ENDM
print-two-char   MACRO   char1, char2   ; 输出两个字符
        mov   AH, 2
        mov   DL, char1
        int     21H
        mov   DL, char2
        int     21H
    ENDM
    .CODE
        ...
        print-two-char   char1, char2
        ...
    .DATA
        char1   Byte   'A'
        char2   Byte   'B'
```

如果传递过多的参数，多余的参数将产生警告（除非你使用 VARARG 保留字）。如果传递过少的参数，编译程序将指派空的字符串给剩余的参数（除非你有指定缺省值），不过这将造成错误的情况发生。例如上例的 writechar 宏没有参数，结果如下：

```
MOV   DL,
```

编译程序在展开语句时会产生错误，但不是宏定义或宏调用的错误。

6.10.4 指定需要或缺省的参数

宏参数可以有特别的属性去使得宏较有弹性和改进错误的处理能力。你可以将必须要有的参数给定他们的缺省值。必须要有的参数可以使用 REQ 伪指令去加强维护。语法如下：

```
parameter: REQ
```

例如，你可以重写 writechar 宏去规定必须要有的参数：

```
writechar   MACRO   char: REQ
    mov   ah, 2
    mov   dl, char
    int     21h
ENDM
```

如果调用没有包含相配的参数，编译程序会报告哪列错误以及参考信息，因此 REQ 可以改进错误报告信息。你也可以指定缺省值去维护缺少的参数。如下：

```
parameter: =textvalue
```

假设你通常使用 writechar 去输出 ASCII 7，下列的宏定义使用一个等号符号去告诉编译程序去假设参数 char 是 7，除非你指定其它值：

```
writechar   MACRO   char: = 〈7〉
    mov   ah, 2
    mov   dl, char
    int    21h
ENDM
```

如果参考此宏时没有包含参数 char，编译程序会用缺省值 7 去填充此空白。注意要用“〈〉”去围住此缺省的参数值。

其实有许多工作可以使用宏或使用子程序（稍后章节会介绍）。要决定使用哪一个，你必须考虑速度与程序码大小。对于重复性的工作，子程序会产生较小的程序码，因为指令实际上只出现在程序中一次。然而每次调用子程序牵涉到 CALL 与 RET 指令的负担，而减缓了程序的执行速度。宏并不需要改变程序的流程，所以执行较快，但会多次产生相同的程序码，而使得程序码较大，其实过大的程序码也会减缓程序执行的速度。

下面我们摘要列举宏重点：

●参数的传递是一对一的，使用时可不要将参数的顺序传错了。

●程序中使用宏，在编译阶段就已完成。实际上编译程序是将宏完整地插入在程序中调用它的地方，就如同我们写的一样，只不过编译程序隐藏了事实，不让我们看见罢了，我们可在 LST 文件看到宏展开的结果。

●程序中调用宏时，若参数传递个数超过宏定义时参数的个数，则多余的参数将被忽略，如 putchar char1，char2，char3... 原始 putchar 宏只有一个参数被定义，char2，char3，... 将被忽略。

● 除非有必要否则不鼓励宏名称用保留字。

●编译程序不在乎输入参数是使用寄存器名称或内存地址，只要宏展开后，合乎汇编语言程序规则就可。我们示范使用宏将 ch6-1.asm 改成 ch6-2.asm，我们可在 ch6-2.lst 看出编译程序在原始文件中如何将宏展开：

```
.DOSSEG          ; ********** CH6_2.ASM **********
.MODEL SMALL ; 示范宏的使用
.STACK 1024
putchar macro char: = 〈  〉
        mov   ah, 2
        mov   dl, char
        int    21h
endm
putstr macro str
       mov   dx, offset str
```

```
        mov ah, 9
        int  21h
endm
clear _kbuffer macro str
        mov dx, offset str
        mov ah OAh
        int 21h
endm
getch macro
        mov  ah, 7
        int   21h
endm


.DATA
        message1   BYTE 'Please input a string→', 0Ah, 0Dh,' $'
        label1     BYTE 80
        label2     BYTE 0
        label3     BYTE 80 dup (' ')
        message2   BYTE "String input OK!",
                        ", Press any key to exit!","$"
.CODE
        MAIN PROC
          .STARTUP
           putstr message1
           mov al, 0ch
           clear _kbuffer label1
           putchar 0A1
           putstr message 2
           getch
          .EXIT
        MAW ENDP
END
```

下面是 CH6 _ 2. LST 文件的内容：

```
Microsoft (R) Macro Assembler Version 6.11          08/15/95 10:41:55
ch6 _ 1.asm                                         Page 1-1


                        .DOSSEG
                        .MODEL SMALL
                        .STACK 1024
```

```
                                    putchar macro char : = 〈  〉
                                            mov ah, 2
                                            mov  dl, char
                                            int   21h
                                    endm
                                    putstr macro str
                                           mov  dx, offset str
                                           mov  ah, 9
                                           int   21h
                                    endm
                                    clear _kbuffer macro str
                                           mov  dx, offset str
                                           mov  ah, 0Ah
                                           int   21h
                                    endm
                                    getch macro
                                           mov  ah, 7
                                           int   21h
                                    endm
0000                                .DATA
0000 50 6C 65 61 73 65                    messagel BYTE 'Please
                                      input a string→', 0Ah, 0Dh,' $'
     20 69 6E 70 75 74
     20 61 20 73 74 72
     69 6E 67 20 2D 3E
     0A 0D 24
001B 50                             label1   BYTE 80
001C 00                             label2 BYTE 0
001D 0050 [                               label3 BYTE 80 dup (' ')
       20
     ]
006D 53 74 72 69 6E 67                          message2 BYTE 'String
                                    input OK!, Press any key to exit!',' $'
     20 69 6E 70 75 74
     20 4F 4B 20 21 20
     2C 20 50 72 65 73
     73 20 61 6E 79 20
     6B 65 79 20 74 6F
     20 65 78 69 74 20
     21 24
0000                                .CODE
0000                                      MAIN PROC
```

```
                                .STARTUP
                                putstr message1
0017  BA 0000 R     1            mov  dx, offset message1
001A  B4 09                 1        mov  ah, 9
001C  CD 21                 1        int  21h
001E  B0 0C                              mov al, 0Ch
                                         clear _ kbuffer label1
0020  BA 001B R     1            mov  dx, offset label1
0023  B4 0A                 1        mov  ah, 0Ah
0025  CD 21                 1        int  21h
                                             putchar 0Ah
0027  B4 02                 1        mov  ah, 2
0029  B2 0A                 1        mov  dl, 0Ah
002B  CD 21                 1        int  21h
                                         putstr message2
002D  BA 006D R     1            mov dx, offset message2
0030  B4 09                 1        mov  ah, 9
0032  CD 21                 1        int  21h
                                       getch
0034  B4 07                 1        mov  ah, 7
0036  CD 21                 1        int  21h
                                        .EXIT
003C                                MAIN ENDP
                                END
Microsoft (R) Macro Assembler Version 6.11          08/15/95 10:41:55
ch6 _ 1.asm                                         Symbols 2-1

Macros:
                    Name                Type

clear-kbuffer . . . . . .           Proc
getch . . . . . .                   Proc
putchar . . . . . .                 Proc
putstr . . . . . .                  Proc

Segments and Groups:

                    Name        Size    Length    Align    Combine Class

DGROUP . . . . . .          GROUP
_DATA . . . . . .           16 Bit  0099    Word    Public 'DATA'
```

```
STACK . . . . . 16 Bit  0400  Para   Stack
'STACK'_TEXT . . . . . 16 Bit  003C  Word   Public 'CODE'
Procedures, parameters and locals:

                                Name          Type  Value  Attr

MAIN . . . . . P Near 0000 _TEXT Length= 003C Public
@Startup . . . . . L Near 0000 _TEXT

Symbols:

                                Name          Type  Value  Attr

@CodeSize . . . . . Number  0000h
@DataSize . . . . . Number  0000h
@Interface . . . . . Number  0000h
@Model . . . . . Number  0002h
@code . . . . . Text  _TEXT
@data . . . . . Text  DGROUP
@fardata? . . . . . Text  FAR_BSS
@fardata . . . . . Text  FAR_DATA
@stack . . . . . Text  DGROUP
label1 . . . . . Byte  001B  _DATA
label2 . . . . . Byte  001C  _DATA
label3 . . . . . Byte  001D  DATA
message1 . . . . . Byte  0000  _DATA
message2 . . . . . Byte  006D  _DATA
        0 Warnings
        0 Errors
```

我们可以看到编译程序先把宏的内容插入程序内宏出现的位置，再进行编译。例如：

```
                        putstr message1
0017  BA  0000 R      1     mov   dx, offset message1
001A  B4  09          1     mov   ah, 9
001C  CD  21          ①     int   21h
```

标有“1”的列就是宏的内容，‘1’是代表第一层的宏，若我们在宏中又使用了前面已定义好的宏（也就是在一个宏中使用了之前已定义好的宏），则就会被标上‘2’，表第二层。

6.11 显示方式（BIOS INT 10H）

前面已介绍过 DOS 基本的屏幕输出、输入服务程序，下面再介绍一个控制屏幕显示的方

法。利用 BIOS 所提供的 INT 10H 屏幕处理服务程序，它比 DOS INT 21H 更有效率、更快，但却没有 DOS 的可靠性高。谈到屏幕影象不得不谈到屏幕显示卡。屏幕显示卡可分单色和彩色。单色显示卡，其文字显色通常是绿色或黄色。彩色显示卡 一般则有 CGA、EGA、VGA。现在大家最常用的是 VGA（Video Graphics Array）。

6.11.1 显示方式

显示方式又可分为文本模式和绘图模式。

如果您有彩色显示卡（CGA、EGA、VGA），您可以调用 INT 10H，将 AL 设成下列各值，去设置各种显示模式。一般屏幕显示模式（图 6-3）介绍如下：

模式（AL）	分辨率	形式	色彩	显示卡（适用）
00H	40 * 25	文字	黑/白	CGA，EGA，VGA，MCGA
01H	40 * 25	文字	16 种彩色	CGA，EGA，VGA，MCGA
02H	80 * 25	文字	黑/白	CGA，EGA，VGA，MCGA
03H	80 * 25	文字	16 种彩色	CGA，EGA，VGA，MCGA
04H	320 * 200	绘图	4 种彩色	MCGA
05H	320 * 200	绘图	黑/白	MCGA
06H	640 * 200	绘图	黑/白	MCGA
07H	80 * 25	文字	黑/白	单色，EGA，VA
08～0AH	—	—	—	PCjr
0DH	320 * 200	绘图	16 种彩色	EGA，VGA
0EH	640 * 200	绘图	16 种彩色	—
0FH	640 * 350	绘图	黑色	—
10H	640 * 350	绘图	64 种彩色	EGA，VGA
11H	640 * 480	绘图	2 种彩色	MCGA，VGA
12H	640 * 480	绘图	16 种彩色	VGA
13H	320 * 200	绘图	256 种彩色	MCGA，VGA

* MCGA（Multi-Color Graphics Array）多色彩图形矩阵。

* 单色显示卡 Monochrome。

图 6-3 屏幕显示模式

6.11.2 显示页

在一般显示卡（如 VGA 卡）上实际安装的 RAM 大小，都比一个屏幕显示所需的内存大。一个屏幕显示所需的内存称为一个 Page（显示页），而一般显示卡有时可提供多个显示页。而显示硬件则可被设置选择其中一个显示页映至屏幕。

由于在一特定时间内，只能显示一个显示页数据至屏幕，因此可以在将数据写入显示页的同时，也将其它数据写至其它非作用的显示页上。在适当时间，将非作用的显示页切换为作用显示页，这个做法可使屏幕更新的速度非常快。

显示页的编号由 0 开始，最多至 7（图 6-4）。设置选择显示页可用 ROM BIOS INT 10H 服务程序的 05H 功能，欲辨别作用中的显示页可用 0FH。理论上，凡不是在作用中的显示页，都可以移作其它用途。但由于这些显示页的用途本来就是做显示画面用，所以最好还是不要移做他用。

显示模式	显示页数	显示卡
00H，01H	8 (0～7)	CGA，EGA，MCGA，VGA
02H，03H	4 (0～3)	CGA
	8 (0～7)	EGA，MCGA，VGA
04H，05H	1 (0)	CGA，MCGA
	2 (0～1)	EGA，VGA
06H	1 (0)	CGA，EGA，MCGA，VGA
07H	1 (0)	Monochrome
	8 (0～7)	EGA，VGA
0DH	8 (0～7)	EGA，VGA
0EH	4 (0～3)	EGA，VGA
0FH	2 (0～1)	EGA，VGA
10H	2 (0～1)	EGA，VGA
11H	1 (0)	MCGA，VGA
12H	1 (0)	VGA
13H	1 (0)	MCGA，VGA

图 6-4　显示页

本节所介绍的 BIOS INT 10H 显示服务程序如下：

AH　服务程序描述	AH　服务程序描述
00H　设置显示方式	07H　将屏幕往下卷
01H　设置光标形状	08H　读取字符和属性
02H　设置光标位置	09H　在光标位置显示字符和属性
03H　读取光标位置	0AH　在光标位置显示字符（不含属性）
05H　设置有效（作用中）的显示页	0FH　取当前显示方式
06H　将屏幕往上卷	

一般来说，INT 10H 会保留 BX、CX、DX 和段寄存器的值，也就是使用 INT 10H 后，这些寄存器的内容并不更动。而其它寄存器若需保留原先的状态，就需自己先将之 PUSH 至 STACK，执行完后再将之 POP 回来。

1. 00H：设置显示方式

AH=0，所选择的模式号码放在 AL。在正常情况下，每次模式设置后，ROM BIOS 会自动清除屏幕内存的缓冲区（即会清屏幕），即使是重复设置相同的模式也不例外。其实，重复设置相同的模式，可能是清除屏幕最简单而又有效的方法。实例如下：

```
mov   AH，0   ；设置显示模式
mov   AL，3   ；选择模式 3，80 * 25 彩色文本方式
int   10H     ；call ROM BIOS
```

若使用的是 CGA、MCGA、VGA 卡，想要达到设置显示方式，而不清除屏幕的效果，只要将所设置的模式号码加上 80H（128），即最高 bit 设为 1，放入 AL 即可。上例只要改为（mov AL，83H）即可。不过在切换不同模式时，不同的显示模式分辨率多少会有差异，所以可能有些数据会被破坏掉，这是不可避免的，设计程序时须注意此点。

2. 01H：设置光标大小

AH＝01H，在正常情况下，光标是以一条或两条闪烁扫描线的形式，显示在字符位置的底部。通常在文字编辑器中，我们可利用此服务程序改变光标的大小，以表示在 Insert 或 Replace 状态，或是隐藏光标。

对 CGA 卡而言，显示光标大小包含 8 条扫描线（图 6-5），编号由最顶部 0 至最底部的 7。其中起始线编号放在 CH，结束线编号放在 CL。对 CGA 而言，缺省的值为 CH＝6、CL＝7。对单色显示卡（MDA）和 EGA 卡而言，可显示光标大小共包含 14 条扫描线，编号由最顶部 0 至最底部 13，缺省值为 CH＝11，CL＝12。MCGA 与 VGA 包含 16 条扫描线，缺省为 CH＝13、CL＝14。

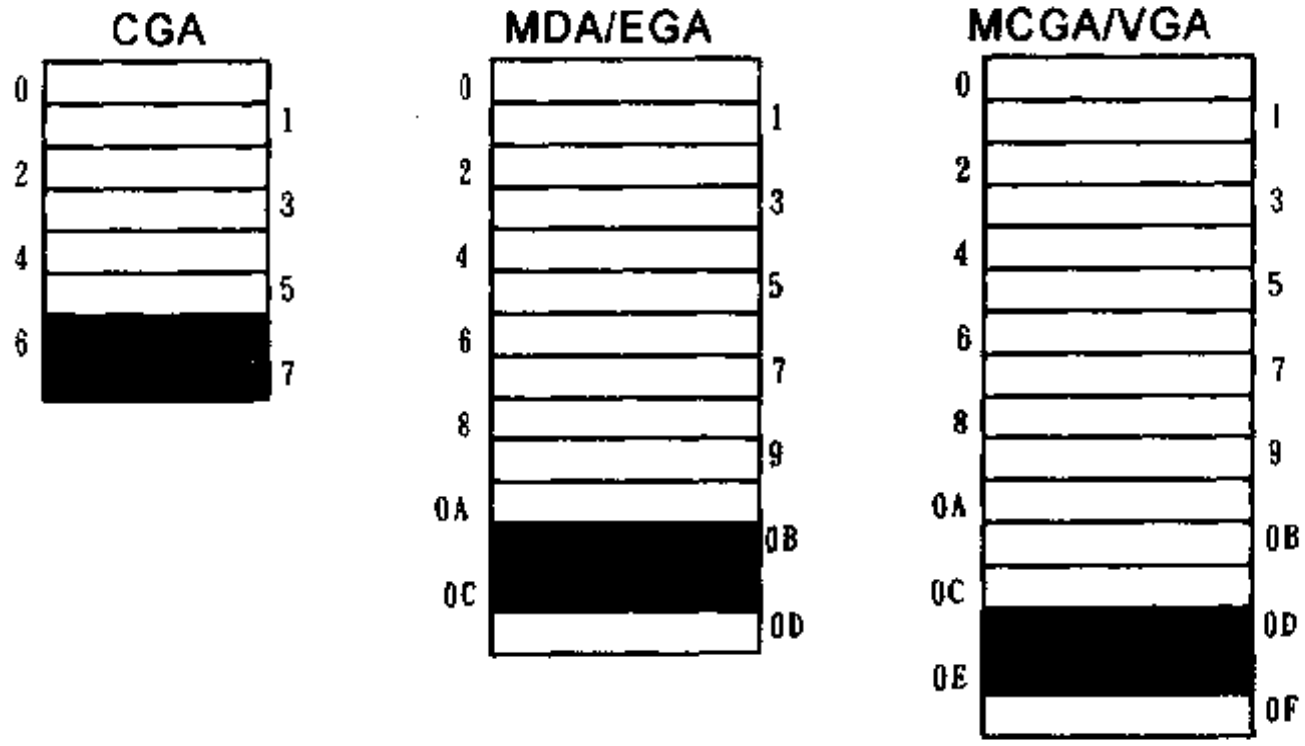


图　6-5

实例如下：

```
mov   AH,   1   ；设置光标大小
mov   CH,   0   ；设置起始线
mov   CL,   7   ；设置终止线
int   10H
```

因为这两个寄存器只用到 4 个 bit（0～3 bit），我们可以利用这个特性来隐藏光标：

（1）把 CH 的位 5 设为 1，即设 CH 为 20H（32），这是在文本方式中移去光标的第一种技巧。

（2）屏幕至多 25 列即（0～24 列），将光标移到屏幕之外，（例如第 25 列第 1 行）。不过程序结束后，记得恢复原态。由于绘图模式没有真正光标，在绘图模式上看到的光标都是利用 DFH 方块字符或改变背景属性来模拟。

3. 02H：设置光标位置

AH＝02H，服务程序 02H 是利用行和列的坐标（也就是 X-Y 坐标），来设置光标位置。列的编号放在 DH，行的编号放在 DL，显示页的编号放在 BH。

在文本方式，可能有好几个显示页，且每个显示页又各自记录其光标位置。绘图模式，

虽然没有可见的光标，但也有一个逻辑光标位置记录，以监管字符的输出、输入位置。列与行编号由屏幕左上角坐标（0，0）开始往右下角排起。要注意，各显示模式有不同显示页数。

实例如下：

```
mov  AH,  2  ; 设置光标位置
mov  DH,  0  ; 设置列号 0（二维坐标的 Y 坐标）
mov  DL,  0  ; 设置行号 0（二维坐标的 X 坐标）
mov  BH,  0  ; 设置编号 0 的显示页
int  10H
```

不同的BIOS 版本，AL 可能在执行完后被改变，可以在执行前先 SAVE，执行完后再回存 AL 值。可利用 STACK 变量来达到此目的。

```
gotoxy  MACRO  x, y
        push  AX  ; 先 save AX 值
        mov   AH, 2
        mov   DH, y
        mov   DL, x
        mov   BH, 0
        int   10H
        pop   AX  ; 再 restore AX 值
ENDM
```

4. 03H：读取光标位置

AH＝03H，把将要读取的显示页编号放在 BH，ROM BIOS 会将光标的状态信息返回在 CX 与 DX 中。

寄存器	意义
CH	光标的起始扫描线编号
CL	光标的结束扫描线编号
DH	光标列的编号
DL	光标行的编号

实例如下：

```
mov  AH,  3          ; 读取光标状态
mov  BH,  0          ; 指定读取第 0 页
int  10H             ; Call BIOS
mov  cursor-size, cx ; 存储光标大小
mov  row,  DH        ; 存储光标列号
mov  col,  DL        ; 存储光标行号
```

在程序中若需打开一个下拉式窗口，光标暂必会被变动。可以利用上述功能在关起下拉式窗口或退出新开的窗口后，回到旧窗口时，回复光标原来的位置及大小。下例会先将光标隐藏起来，最后再将光标回复原值：

```
mov  AH,  3                ; 读取目前光标状态
```

```
mov   BH,    0              ; 假设第 0 页
int   10H
mov   cursor-size, CX       ; 先记录光标大小在 cursor-size
mov   cursor-position, DX   ; 先记录光标位置在 cursor-position
mov   AH,    1
mov   CH, 00100000B         ; 将 bit 5 设为 1，以隐藏光标
int   10H
...
mov   AH,    2              ; 回复光标位置
mov   DX, cursor-position
mov   BH, 0
int   10H
int   10H
mov   AH, 1                 ; 回复光标大小
mov   CX, cursor-size
int   10H
```

5. 05H：设置有效显示页

AH＝05H，将选择显示的有效显示页放入 AL 中。此功能可作用在 CGA、EGA、VGA 卡的文本方式 0～3、16 种颜色的绘图模式。

在 80＊25 文本方式下，CGA 卡只提供四个显式页（0～3），所以选择第 4～7 页是与 0～3 页重复的，即选择第 4 页与第 0 页的内容相同。

所有显示模式，第 0 页都是缺省的显示页，而且，第 0 页都是位于显示内存的开头（即较低的地址），页数越高的显示页，则位于较高的内存位置（图 6-6）。

实例如下：

```
mov   AH,    5              ; 设置有效显示页
mov   AL,    0              ; 选择第 0 页为有效显示页
int   10H                   ; Call BIOS
```

显示模式	起始节地址	一个显示页所需的内存大小（bytes）	显示卡
00H，01H	B800H	2000	CGA EGA MCGA VGA
02H，03H	B800H	4000	CGA EGA MCGA VGA
04H，05H	B800H	16000	CGA EGA MCGA VGA
06H	B800H	16000	CGA EGA MCGA VGA
07H	B000H	4000	MDA EGA VGA
0DH	A000H	32000	EGA VGA
0EH	A000H	64000	EGA VGA
0FH	A000H	56000	EGA VGA
10H	A000H	112000	EGA VGA
11H	A000H	38400	MCCGA VGA
12H	A000H	153600	VGA
13H	A000H	64000	MCGGA VGA

图　6-6

页数	40×25（16色）	80×25（16色）	80×25（单色）
0	B800：0000H	B800：0000H	B000：0000H
1	B800：0800H	B800：1000H	B000：1000H *
2	B800：1000H	B800：2000H	B000：2000H *
3	B800：1800H	B800：3000H	B000：3000H *
4	B800：2000H	B800：4000H *	B000：4000H *
5	B800：2800H	B800：5000H *	B000：5000H *
6	B800：3000H	B800：6000H *	B000：6000H *
7	B800：3800H	B800：7000H *	B000：7000H *

* 只适用 EGA 及 VGA

* 40×25（16色）：显示模式 00H，01H

* 80×25（16色）：显示模式 02H，03H

* 80×25（单色）：显示模式 07H

图 6-7 CGA，MCGA，EGA，VGA 文字模式下显示页的起始地址

6.11.3 文本方式的字符显示

文本方式中，屏幕上每一个位置的字符皆为内存中两个相邻的字节所控制。第一个字节为被显示字符的 ASCII 码，第二个字节为此字符的属性也就是控制此字符显示的方式。

利用 ASCII 码的索引，显示卡上的字符产生器便可将该字符画在屏幕上。对单色显示卡而言，一个字符格式为 9 * 14 个象素。对彩色显示卡而言，字符格式为 8 * 8 个象素。由于单色显示卡的字符格式较大，所以较易于阅读。

在文本方式，所有字符中，有一半是标准的 ASCII 字符（10H 至 7FH）。0 为 NULL 字符，显示为空的，什么都看不见。另一半为 128 个（80H～0FFH）图形字符。其中有许多字符可用来绘制简单的直线图。

6.11.4 文本方式的字符对映方式

02H、03H、07H 的显示模式皆为文本方式 80 * 25。由于每个字符需 2 个 Byte 存储，所以一个屏幕便需占用 80 * 25 * 2＝4000 Bytes，大约为 4KB。若使用 EGA 或 VGA 显示卡则有 16KB 的显示用内存（文本方式下），共能划分成 16/4＝4 个显示页。

屏幕左上角第 1 个字符（0，0）所占用的内存地址为：

B800：0000（ASCII），B800：0001（属性）。

由于文本方式显示页地址皆由“偶数的 K”开始，如：

B800：0800→B800：1000

B800：1000→B800：2000

所以在每一个显示页之后，都会留下一些未用的位置元组：

4K－80 * 25 * 2＝96 Bytes（40 * 25）

2K－40 * 25 * 2＝48 Bytes（80 * 25）

所以在计算任何显示页屏幕上字符的地址时，须小心计算：

①40 * 25 格式（列 0～24，行 0～39）

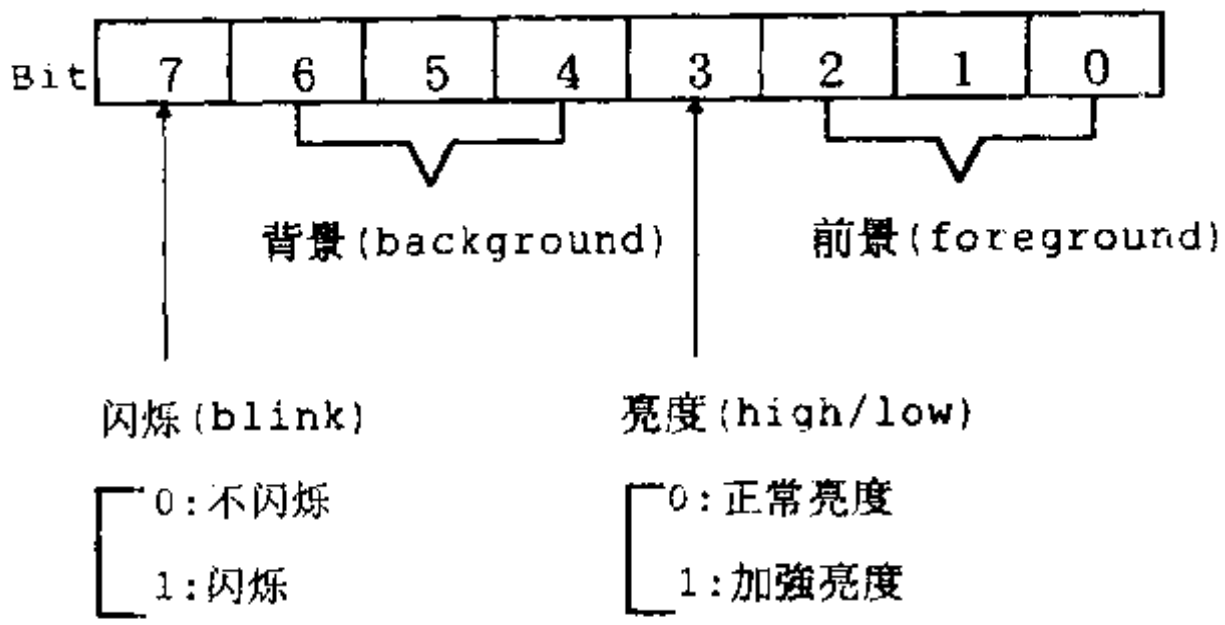
字符偏移＝B800：0000＋（列 * 40 * 2＋行 * 2）

②80 * 25 格式（列 0～24，行 0～79）

字符偏移＝B800：0000＋（列＊80＊2＋行＊2）

6.11.5 文本方式的字符属性

文本方式中每个位置字符的第2字节为属性，是控制第1字节所代表ASCII码字符的色彩、亮度和闪烁的特性。



属性值(Hex)	属性说明
00H	不显示
01H	底线
07H	正常(黑底白字)
09H	高亮度加底线
0FH	高亮度
70H	反白(白底黑字)
87H	闪烁(黑底白字)
8FH	高亮度加闪烁
F0H	反白加闪烁

图6-8 单色屏幕文本方式所能显示的属性

1．06H、07H：将窗口往上卷或往下卷

AH＝06H或07H，服务程序06H或07H，会将整个文字窗口的内容往上或往下卷动一列或多列。服务程序06H（往上卷），把窗口顶部数据卷出窗口之外，并把空白列插入窗口顶部。服务程序07H（往下卷），把窗口底部数据卷出窗口之外，并把空白列插入窗口顶部。卷出的数据均无法复原。

前景或背景	色彩	前景	色彩
0000	black	1000	gray
0001	blue	1010	light blue
0010	green	1011	light green
0011	cyan	1011	light cyan
0100	red	1100	light red
0101	magenta	1101	light magenta
0110	brown	1110	yellow
0111	white	1111	bright white

图6-9 彩色屏幕文本方式所能显示的属性

```
    mov   AH，06    ；将窗口往上卷
    mov   CX，0     ；左上角起始位置(0，0)
    mov   DX，184FH ；右下角终止位置(24，79)
                    04H
    mov   BH，07H   ；空白列的显示属性
    int   10H，     ；Call BIOS
```

上例会达到将整个窗口清除的效果。

服务程序编号	所需寄存器值
AH＝06H（往上卷动） AH＝07H（往上卷动）	AL＝欲卷动的列数 BH＝空白列的显示属性 CH＝左上角的列号 CL＝左上角的行号 DH＝右下角的列号 DL＝右下角的行号

若将 AL 设为 0 或大于窗口所能显示的最大列数，也可以达到将整个屏幕清除的效果。一般我们卷动窗口可分为两个步骤：

（1）调用服务程序 06H 或 07H，卷动欲清除的窗口内容。

（2）再调用服务程序 02H，设置光标位置及写入字符服务程序，将新数据写入窗口中。

2. 08H：读取光标所在位置字符和属性

AH＝08H，服务程序 08H，是用来读取在屏幕上的字符和属性，也就是读取显示内存内的字符和属性，且在文字及绘图模式下皆能读取。

先在 BH 设置欲读取的有效显示页，BIOS 会将光标所在位置字符 ASCII 码放在 AL 中。在文本方式下，还会将属性返回到 AH。

服务程序编号	所需的寄存器值	返　回　值
AH＝08H	BH＝有效显示页	AL＝光标所在位置的 ASCII 字符码。 AH＝文字字符的属性（文本方式下）。

下列实例示范指定读取列＝1，行＝2 位置的字符和属性：

```
mov   AH，2          ；设置光标位置
mov   BH，0
mov   DX，0102H      ；光标位置（1，2）
int   10H
mov   AH，8          ；读取字符和属性
mov   BH，0
int   10H
mov   char，AL       ；存储字符
mov   attribute，AH  ；存储属性
```

3. 09H：在光标位置显示字符和属性

AH＝09H，服务程序 09H 可用来写出一个或多个相同的字符及属性。欲写出的字符放在 AL 中，属性放在 BL 中，写出的次数放在 CX 中，BH 为存放的显示页。

服务程序编号	所需的寄存器值
AH＝09H	AL＝写出的 ASCII 字符 BL＝在光标位置显示字符的属性（文本方式）前景颜色（绘图模式） BH＝显示页 CX＝在光标位置显示字符和属性的次数

使用实例如下：

```
mov   AH，9      ；在光标位置显示字符和属性
mov   AL，'A'    ；写出'A'
mov   BH，0      ；第 0 页
mov   BL，70H    ；以反白属性写出'A'
```

```
mov   CX，10     ；重复写'A'10次
int   10H
```

CX虽可指定重复写出相同字符多次，但光标并不会跟着移动，还是停在光标原先的位置，即第1个字符的位置。

若重复的次数超过本列所能显示的字符数，则会继续在下列输出。如遇屏幕最底部那一列（第25列），会停止在光标位置显示字符，即屏幕不会自动卷动。

若写出的ASCII字符为0～20H的控制字符，则会输出相对的图形字符，并不会像INT 21H DOS服务程序02H，解释为控制字符。

4. 0AH：在光标位置显示字符

AH=0AH，与服务程序09H相似，只是不能指定颜色属性。

服务程序编号	所需寄存器值
AH=0AH	AL=欲写出的ASCII字符 BL=前景颜色（绘图模式下） BH=显示页 CX=写出次数

使用实例如下：

```
mov   AH，0AH     ；在光标位置显示字符
mov   AL,'%'      ；指定字符'%'
mov   BH，0       ；第0页
mov   CX，1       ；写一次
int   10H
```

5. 0FH：取当前显示方式

AH=0FH，服务程序0FH，可用来返回目前屏幕的显示模式在AL，屏幕宽度（80或40行）在AH，和显示页的编号在BH中。使用实例如下：

```
mov   AH，0FH
int   10H
mov   video-mode，AL
mov   video-page，bH
mov   screen-width
```

※ 在要切换目前显示模式去做其它显示模式输出时，利用此功能可将原先的显示模式先存储起来，以便回到原显示模式。

下面CH6-3.ASM程序示范了INT 10H的许多基本功能。我们均将各种功能写成相对的宏，使读者能多了解宏的使用，当然也增加了程序的可读性。不然一长串的指令各代表何目的，会让人眼花瞭乱。

程序一开始先把原先的光标的属性记录在变量(cursor-attrib)，利用卷动屏幕的功能达到清除屏幕的目的。接下来再利用卷动屏幕功能，设置背景为蓝色，前景为黄色，开一个新窗口，并在新窗口底部最后一列显示出信息，等待用户按键后再退出，并清除屏幕，恢复原来的光标属性。

CH6 _ 3. ASM

```
.DOSSEG
.MODEL small
.STACK 1024
CLS     MACRO
        mov  ah, 7
        mov  al, 0
        mov  bh, 07h
        mov  cx, 0
        mov  dx, 184fh
        int  10h
ENDM
SCROLL _ UP  macro line, attrib, left _ up, right _ down
        mov  ah, 6
        mov  al, line
        mov  bh, attrib
        mov  cx, left _ up          mov  dx, right _ down
                                    int  10h
ENDM
SET _ CURSOR MACRO position
          mov  ah, 2
          mov  bx, 0
          mov  dx, position
          int  10h
ENDM
CURSOR _SIZE MACRO cursor _ line
          mov  ah, 1
          mov  cx, cursor _ line
          int  10h
ENDM
READ _ CURSOR MACRO
          mov  ah, 3
          mov  bh, 3
          int  10h
ENDM
PUTSTR MACRO str
          mov  dx, offset str
          mov  ah, 9
          int  21h
ENDM
.DATA
        message1        BYPE 'Press any key to quit!  $'
```

```
        cursor _ attrib WORD?

.CODE
        GETCHAR MACRO
            mov   ah, 7
            int    21h
        ENDM
        .STARTUP
          read _ cursor
          mov cursor _ attrib, cx
          cls
          scroll _ up 11, 1Eh, 514h, 0F3Ch
          set _ cursor 0E1Ah
          putstr message1
          cursor _ size 2000h
          getchar                    ; GETCHAR 宏定义要写在此列之前。
          cls                        ; 切记! 切记!
          cursor _ size cursor _ attrib
        .EXIT
END
```

各位应该知道要破坏屏幕的原貌之前应先将原来屏幕相关的信息存储起来，使之能恢复原貌（如果要复原的话）。此程序示范了如何恢复光标属性。在清屏幕 CLS 宏中，使用了 07H 功能，其实也可以使用 06H，达到相同的效果。

有没有在代码段中，看到 GETCHAR 宏，它示范了宏可定义在任何地方。不过有一个限制要记住，就是要写在调用它之前。

6.12 INCLUDE 指令

前面我们建立了许多常用的宏，但在我们秉持让程序多做点，人少做点的原则下，每次使用宏都要再重写一次宏定义，还是太麻烦了。为了让我们能减轻程序设计的负担，接下来我们将介绍 INCLUDE 伪指令的使用。

只要将常用的宏全部写在一起，存储在同一个文件中，在程序中利用 INCLUDE 伪指令将此文件内容含括进来即可。我们举例说明较实际。

举 CH6 _ 3. ASM 为例。程序定义了 7 个宏——CLS、SCROLL _ UP、SET _ CURSOR、CURSOR _ SIZE、READ _ CURSOR、PUTSTR、GETCHAR。我们将此 6 个宏存储在一个叫 CH6 _ 3. INC 的文件中。CH6 _ 3. INC 的内容如下：

```
CLS     MACRO
        mov   ah, 7
        mov   al, 0
        mov   bh, 07h
```

```
        mov   cx,  0
        mov   dx,  184fh
        int     10h
ENDM

SCROLL _ UP MACRO line, attrib, left _ up, right _ down
          mov   ah, 6
          mov   al, line
          mov   bh, attrib
          mov   cx, left _ up
          mov   dx, right _ down
          int     10h
ENDM
SET _ CURSOR MACRO position
          mov   ah, 2
          mov   bx, 0
          mov   dx, position
          int     10h
ENDM

CURSOR _ SIZE MACRO cursor _ line
          mov   ah, 1          mov   cx, cursor-line
                               int     10hENDM

. READ _ CURSOR MACRO
           mov   ah, 3
           mov   bh, 3
           int     10h
ENDM
PUTSTR MACRO str
          mov   dx, offset str
          mov   ah, 9
          int     21h
ENDM
```

我们将 CH6 _ 3. ASM 改为 CH6 _ 4. ASM 的内容。其实两文件的内容都相同，只是CH6 _ 4. ASM 少了 6 个宏的定义，而改而 INCLUDE CH6 _ 4. INC，执行的情况完全相同，目的只是示范 INCLUDE 的使用而已。当然你也可以将定义在程序区里的 GETCHAR 宏也写在 CH6 _ 4. INC 中。

CH6 _ 4. ASM

```
. DOSSEG
. MODEL small
. STACK 1024
INCLUDE CH6 _ 3. INC
. DATA
        messagel      BYTE 'Press any key to quit ! $'
        cursor _ attrib WORD?
. CODE
        GETCHAR MACRO
            mov  ah, 7
            int   21h
        ENDM

        . STARTUP
        read _ cursor
        mov cursor _ attrib, cx
        cls
        scroll _ up 11, 1Eh, 514h. 0F3Ch
        set _ cursor 0E1Ah
        putstr messagel
        cursor _ size 2000h
        getchar                    ; GETCHAR 宏定义要写在此列之前。
        cls                        ; 切记! 切记!
        cursor _ size cursor _ attrib
        . EXIT
END
```

其实，CH6 _ 4. INC 的文件名可以自行定义，不过副文件名习惯用 . INC 较具意义。INCLUDE 在程序的位置习惯插在代码段之前，插入的位置并没有一定的限制，你可以随意在任何位置使用。有个原则一定要把握，一定要在使用宏之前插入，还有就是要在 END 之前，不然编译程序便找不到它了，记住编译程序只编译在 END 之前的程序，之后你写什么没人管你，编译程序当然更不会理你。

第 7 章　算术运算指令

任何计算机语言程序，很难避免使用到算术运算，Intel 指令集提供 8 位、16 位和 32 位的整数（integer）运算指令。

在这一章节，我们将介绍一些基本的算术运算指令，读者可通过本章了解汇编语言加、减、乘、除四则运算的用法，逻辑及位移循环移位指令。这一章节解释了适于 8086、8088、80286、80386 和 80486 处理器的指令格式。对于所有处理器所共用的标志寄存器说明如图 7-1 所示。指令运算完毕后，可能会改变在标志内某个相对的值。如运算结果产生溢位，则 OF 标志值将被设为 1，这个操作是由处理器自动完成。我们可以利用某个测试标志值的指令去判断运算的结果是否正常，当作下一个程序流程的参考。所以标志值也是在程序设计过程中需要考虑的重点之一，尤其在算术运算的场合，更显得不可缺少。当然也可以根据需要自己动手去设置标志值。

缩写	标　志
O	Overflow
D	Direction
I	Interrupt
T	Trap
S	Sign
Z	Zero
A	Auxiliary carry
P	Parity
C	Carry

图 7-1　标志寄存器说明

值	影　响
1	设置标志
0	清除标志
?	可能改变标志值，但不肯定
空白	不影响
±	根据情况去设置标志值

图 7-2　说明标志改变意义

7.1　加法运算指令

指令格式	O	D	I	T	S	Z	A	P	C
ADD　目的操作数，源操作数	±				±	±	±	±	±
ADC　目的操作数，源操作数	±				±	±	±	±	±
INC　目的操作数	±				±	±	±	±	

7.1.1　ADD、ADC 和 INC 指令

ADD 与 INC 指令，在 8086～80286 上可执行 8 位和 16 位的加法运算，在 80386/486 上可执行 8 位、16 位和 32 位的加法运算。与 ADC 指令结合在一起使用，在 8086 上可处理 32 位的值，在 80386/486 上可处理 64 位的值。

ADD　目的操作数，源操作数

ADC　目的操作数，源操作数

INC　目的操作数

执行 INC 指令，目的操作数本身会累加 1（与 C 的＋＋是相同的意思）。目的操作数可以是 8 位、16 位、或 32 位的内存变量以及除了段寄存器以外的所有寄存器。INC 指令会将整数视为无正负号的值，所以当有进位或借位时也不会改变进位标志。也就是说，INC 指令是一个无正负号的加法指令，除了进位标志外，其它状态标志都会受影响。

ADD 和 ADC 指令可执行两个 8 位、两个 16 位操作数的加法运算。执行完后，源操作数的内容并不改变。不同的是，ADC 指令执行时会顺便把进位标志（CF）的值也加入目的操作数。有两个限制：

（1）两个操作数中，只能有一个是内存操作数（变量），所有段寄存器不能当作目的操作数，所有状态标志都会受影响。

（2）两个操作数必须是相同大小。

相同的指令而有不同种类的操作数在相同的处理器上执行时会有不同的时钟周期（Clock Cycles）。就算是相同的操作数，不同的处理器也会有不同的时钟周期。可以说较新的处理器拥有较少的时钟周期，才能执行得更快。

·ADD 指令实例如下：

```
ADD  AL,  2       ; 立即数 2 加到 8 位寄存器中
ADD  BL,  AL      ; 8 位寄存器相加
ADD  AX,  1000h   ; 立即数 1000h 加到 16 位寄存器中
ADD  var1,  DX    ; 16 位寄存器加到 16 位变量中
ADD  var1,  10    ; 立即数 10 加到 16 位变量中
ADD  CX,  var1    ; 把 16 位变量加到 16 位寄存器中
```

·ADC 指令实例如下：

```
ADC  AX,  DX      ; 将 DX 和进位标志加到 AX 中
ADC  AL,  var1    ; 将 8 位变量和进位标志加到 AL 中
ADC  Var2,  10    ; 将立即数 10 和进位标志加到 16 位变量中
ADC  AX,  10      ; 将立即数 10 和进位标志加到 16 位寄存器中
```

·INC 指令实例如下：

```
INC  AX                  ; 16 位寄存器加 1
INC  Var1                ; 8 位或 16 位变量加 1
INC  Table [bx] +3 ; 将 BX+3 所指定的内存操作数加 1
INC  byte ptr Var1   ; 8 位变量加 1
INC  word ptr Var2   ; 16 位变量加 1
```

一般执行加 1 的运算时，INC 指令的执行速度比 ADD 指令更快。

7.1.2 ADD 和 ADC 对标志的影响

若执行结果目的操作数的最高位有进位产生，则 CF＝1，否则 CF＝0。

```
MOV  BH, 10h
MOV  AL, 0F0h
ADD  AL, BH      ; AL=00h, CF=1, BH=10h
ADC  BH, 5       ; BH=16h, CF=0
```

若执行结果为零，则 ZF＝1，否则 ZF＝0。

```
MOV   BH,    VAR1   ; BH=-2
ADD   BH,    VAR2   ; BH=0, ZF=0
;
var1  byte   -2
var2  byte   2
```

若执行结果超出有符号数所能存储的最大值，则溢位标志 OF=1，否则 OF=0。

```
MOV   BX,    +32767
ADD   BX,    2      ;   BX=-32767, (8001H), OF=1
```

你不可以使用 ADD value1，value2

如果 value1 与 value2 是两个变量（相同大小），你必须先将第一个操作数 copy 到一个寄存器中，如下：

```
MOV   AX, value2
ADD   value1,   AX
```

CH7.1.ASM 首先清除屏幕，然后输出‘1+2+3=’的字符串，利用 ADD 指令将 BX 指针所指到的连续三个内存操作数相加总和输出。要记住，原本总和应是 6，但要输出必须转换成 ASCII 字符‘6’才可正确输出。可别忘了加 30h（48），而直接将数字 6 当作输出。ASCII 字符‘0’是 30h（48），‘1’是 31h（49），所以若要输出‘6’必须加上 30h。

CH7_1.ASM

```
.DOSSEG
.286
.MODEL SMALL
.STACK 1024
 CLS    macro
        mov  ah, 7
        mov  al, 0
        mov  bh, 07h
        mov  cx, 0
        mov  dx, 184fh
        int   10h
endm
PUTSTR  macro str
        mov  dx, offset str
        mov  ah, 9
        int   21h
endm
PUTCHAR  macro char
         mov  ah, 2
         int   21h
endm
```

```
.DATA
        message   BYTE '1-2+3= $'
        array     BYTE 1, 2, 3

.CODE
        main proc
                .STARTUP
                ;
                cls
                PUTSTR message
                mov bx, offset array
                mov, al, [bx]
                add al, [bx+1]
                add al, [bx+2]
                add al, 30h
                putchar al
                ;
                .EXIT
        main endp
END
```

7.1.3 INC 对标志的影响

即使执行结果大于操作数所能存储的最大值，CF 也不会被设为 1。

MOV　BL，255

INC　BL　；BL＝0，CF＝0

若执行结果为零，则 ZF＝1，否则 ZF＝0。

MOV　BH，　－1

INC　　BH　；BH＝0，ZF＝1

7.2　减法运算指令

指令格式	O	D	I	T	S	Z	A	P	C
SUB　目的操作数，源操作数	±				±	±	±	±	±
SBB　目的操作数，源操作数	±				±	±	±	±	±
DEC　目的操作数	±				±	±	±	±	
NEG　目的操作数	±				±	±	±	±	±

7.2.1　SUB、SBB 和 DEC 指令

SUB 与 DEC 指令，在 8086～80286 上可执行 8 位和 16 位的减法运算，在 80386/486 上可执行 8 位、16 位和 32 位的减法运算。与 SBB 指令结合在一起使用，在 8086 上可处理 32 位的值，在 80386/486 上可处理 64 位的值。

SUB　目的操作数，源操作数

SBB　目的操作数，源操作数

DEC　目的操作数

执行 DEC 指令，目的操作数本身会累减 1（与 C 语言的--是相同的意思）。目的操作数可以是 8 位、16 位或 32 位的内存变量以及除了段寄存器以外的所有寄存器。DEC 指令会将整数视为无正负号的值，所以当有进位或借位时也不会改变进位标志。也就是说，DEC 指令是一个无正负号的减法指令，除了进位标志外，其它状态标志都会受影响。

SUB 和 SBB 指令可执行两个 8 位、两个 16 位操作数的加法运算。执行完后源操作数的内容并不改变。不同的是，SBB 指令执行时多了减进位标志的操作。有两个限制：

（1）两个操作数中，只能有一个是内存操作数（变量），所有段寄存器不能当做目的操作数，否则所有状态标志都会受影响。

（2）两个操作数必须是相同大小。

·SUB 指令实例如下：

```
SUB   DL,     2        ; 8 位寄存器减去立即数 2
SUB   CL,     DL       ; 8 位寄存器相减
SUB   BX,     1000h    ; 16 位寄存器减去立即数 1000h
SUB   Var1,   BX       ; 16 位内存操作数减去 16 位寄存器
SUB   Var2,   10h      ; 内存操作数减去立即数 10h
```

·SBB 指令实例如下：

```
SBB   DX,     20       ; 16 位寄存器减立即数和进位标志
SBB   DH,     DL       ; 8 位寄存器相减并减进位标志
SBB   AX,     10       ; 16 位寄存器减立即数 10 与进位标志
SBB   Var1,   10       ; 16 位变量减立即数与进位标志
```

·DEC 指令实例如下：

```
DEC   AL               ; 8 位寄存器减 1
DEC   SI               ; 16 位寄存器减 1
DEC   Var1             ; 8 位变量减 1
DEC   byte ptr Var2    ; 8 位变量减 1
DEC   word ptr Var3    ; 16 位变量减 1
```

一般执行减 1 运算时，DEC 指令的执行速度比 SUB 指令的执行速度要快。

1. SUB 和 SBB 对标志的影响

若执行结果目的操作数的最高位产生借位（Borrow），则 CF＝1，否则 CF＝0。

```
MOV   AX,    0304h
SUB   AL,    5   ; AL＝0FFh，CF＝1，SF＝1
SBB   AH,    3   ; AH＝AH－CF－3→AH＝0FFh
```

若减法结果使得目的操作数结果为零，则 ZF＝1，否则 ZF＝0。

```
MOV   AX，060h
SUB   AX，var1   ; AX＝0，ZF＝1
;
```

var1 WORD 60h

若减法结果超出有符号数所能表示的最小值时，溢位标志 OF＝1，否则 OF＝0。

```
MOV   BX,   -32766
SUB   BX,   3   ; BX=32767D (7FFFh), OF=1
                ; 结果 BX 为正数
```

下面实例 CH7\2.ASM 示范 8 位的有符号与无符号数加法和减法。

```
TITLE 8-BIT signed and unsigned addition and subtraction

.DOSSEG
.MODEL SMALL
.STACK 1024

.DATA
    mem8 BYTE 39
.CODE
    .STARTUP
; Addition
                    ;                       signed unsigned
  mov  al, 26       ; Start with register    26      26
  inc  al           ; Increment               1       1
  add  al, 76       ; Add immediate         +76     +76
                    ;                       ----    ----
                    ;                        103     103
  add  al, mem8     ;                       +39     +39
                    ;                       ----    ----
  mov  ah, al       ; Copy to AH            -114     142
                    ;                  overflow, OF=1
  add  al, ah       ; Add register                   142
                    ;                                ----
                    ;                           Carry, CF=1

; Subtraction
                    ;                       signed unsigned
  mov  al, 95       ; Load register          95      95
  dec  al           ; Decrement              --1     -1
  sub  al, 23       ;                       -23     -23
                    ;                       ----    ----
                    ;                        71      71
  sub  al, mem8     ;                      -122    -122
                    ;                       ----    ----
                    ;                       -51     205
                    ;                             SF=1
```

```
mov   ah, 119        ; Load register          119
sub   al, ah         ;    and subtract        -51
                     ;                        .....
                     ;                          86, OF=1
                     ;                        overflow
    .EXIT
END
```

INC 与 DEC 指令一律将操作数视为无符号数，也就是没有有符号数的运算。就算是我们将此操作数解释为有符号数，在运算的过程中会产生进位或借位，也不会自动去修改设置进位标志值（CF）。

当 8 位的操作数运算结果超过 127，处理器会设置溢位标志（OF=1）。(如果两个操作数都是负数且运算的结果小于或等于-128，溢位标志也会设为 1)。你可以在程序中放置一个 JO（Jump on Overflow）或 INTO（Interrupt on Overflow）指令，转移程序流程至一个错误处理地方。当运算结果超过 255，处理器会设置进位标志（CF=1)。你可以利用 JC 指令去决定程序的流程。

在 CH7-2. ASM 中，如果运算结果小于零，处理器会设置正负号标志（SF=1)。在这个地方可以利用 JS 指令转移程序流程至一个错误处理地方。JO、JC 与 JS 指令在稍候的章节中会介绍。

2. 32 位的加法与减法

当你加减操作数超过一个寄存器所能存储的大小时，如操作数超过 16 位寄存器大小（在 16 位处理器上使用超过 65535)，可利用两个寄存器（DX：AX）去存储 4 bytes 大小的操作数并利用 ADC 和 SBB 指令去处理。当然要记住 DX 存储较高的 word，AX 存储较低的 word。

下面的实例示范 32 位的加法。如果你是在 80386/486 处理器运算，可以利用下面的技巧去处理 64 位的运算：

```
        .DATA
mem32   DWORD  316423
mem32a  DWORD  316423
mem32b  DWORD  156739
        .CODE
        :
        :
; Addition
        mov   ax, 43981                    ; Load immediate
        sub   dx, dx                       ;    into DX: AX
        ; 主要是要让 DX=0，因为并未超过 AX 所能存储的大小 65535
        ; 所以存放在 AX 即可。当然也可以使用 mov dx, 0
        add ax, WORD PTR mem32 [0]         ; Add to both
        adc dx, WORD PTR mem32 [2]         ; memory words
                                           ; Result in DX: AX
```

```
; Subtraction
        mov   ax, WORD PTR mem32a [0]        ; Load mem32
        mov   dx, WORD PTR mem32a [2]        ;    into DX: AX
        sub   ax, WORD PTR mem32b [0]        ; Subtraction low
        sub   dx, WORD PTR mem32b [2]        ;    then high
                                             ; Result in DX: AX
```

如果是在 80386/486 处理器上使用 32 位寄存器，只需要两个步骤即可。若又要考虑到可以在不同的处理器上编译，可以使用 IF 条件语句编译，实例如下：

```
        .DATA
mem32   DWORD  316423
mem32a  DWORD  316423
mem32b  DWORD  156739
p386    TEXTEQU (@Cpu AND 08h)
        .CODE
        :
; Addition
        IF p386
        mov  eax, 43981      ; Load immediate
        add  eax, mem32      ; Result in EAX
        ELSE
                ; 执行先前实例所使用的步骤
        ENDIF
; Subtraction
        IF  P386
        mov  eax, mem32a     ; Load memory
        sub  eax, mem32b     ; Result in AX
        ELSE
                ; 执行先前实例所使用的步骤
        ENDIF
```

在这个实例中你也可以使用 ADC 与 SBB 指令代替 ADD 与 SUB 指令，不过之前要先确定 CF＝0（可使用 CLC；Clear Carry Flag 清除进位标志使其为零。

7.2.2 NEG 指令

在一些有符号数的基本运算中，你可以使用 NEG 指令将有符号数的正负号位改变，也就是将正整数变为负整数，负整数变为正整数。

NEG　目的操作数

目的操作数可以是一个寄存器或内存变量。运算的方式是以零去减操作数的值。若操作数的值为零，则运算结果当然为零，且 CF＝0。否则在其余的情况下，CF 必须设为 1。若目的操作数为－128（8 位）或－32768（16 位）时，因为并无对应的正数（只到＋127 或＋32767），因此－128 或－32768 NEG 的结果还是－128 或－32768，且 OF＝1，CF＝1。

NEG 指令实例如下：

```
NEG   AX
NEG   value
NEG   SI
NEG   CL
```

7.3 乘法运算指令

指令格式		O	D	I	T	S	Z	A	P	C
MUL	源操作数	±				?	?	?	?	?
IMUL	源操作数	±				?	?	?	?	±

8086 家族的处理器对于有符号数与无符号数使用不同的乘法及除法指令，而且这些指令也可依照操作数的大小而有不同的执行操作。描述如下：

MUL 及 IMUL 指令

MUL 指令适用于无符号数的乘法，IMUL 指令适用于有符号数的乘法。

```
MUL    源操作数
IMUL   源操作数
```

MUL 和 IMUL 指令对于 8 位的操作数乘法运算都是假设以寄存器 AL 为目的操作数，16 位以寄存器 AX 为目的操作数，32 位以寄存器 EAX 为目的操作数。如果源操作数为 8 位，则将源操作数乘以 AL 并把 16 位结果存放在 AX。如果源操作数为 16 位，则将源操作数乘以 AX 并把 32 位结果存放在 DX：AX（DX 存放高 word，AX 存放低 word）。如果源操作数为 32 位（80386/486），则将源操作数乘以 EAX 并把 64 位结果存放在 EDX：EAX 中。

在执行 MUL 及 IMUL 指令时，AX 或 DX：AX 固定作为目的操作数，而源操作数可作为内存操作数或寄存器，要注意不能以立即数当做源操作数。

MUL 指令实例如下：

(1) AL 乘以 20h

```
MOV   AL,    5h
MOV   BL,    20h
MUL   BL              ; AX=00A0h, BL=20h, DX=?
```

(2) AX 乘以 20h

```
MOV   ax,    Var1
MUL   Var2            ; DX=0004h, AX=0000h
 ⋮
Var1   WORD   2000h
Var2   WORD   0020h
```

(3) NUM1 乘以 NUM2，并且将结果存放在变量 result 内。

```
MOV   BX, NUM1
MOV   AX, NUM2
MUL   BX,             ; DX: AX=0004 : 48Cch
```

```
; 将 AX 存放在 result 的 low word 中，DX 存放在 high word 中
  MOV  WORD  PTR  result,   AX
  MOV  WORD  PTR  result+2,   DX
  .DATA
        NUM1   WORD   20H
        NUM2   WORD   2466h
        result   DWORD ?
```

IMUL 指令实例如下：

```
MOV    AX,   10
MOV    CX, -48
IMUL   CX                    ; DX: AX=FFFF : FE20h (-480)
MOV    RESULT, AX
MOV    RESULT+2, DX    ; 可写成 MOV RESULT [2], DX
:
RESLUT  WORD  0, 0      ; RESULT DWORD 0 也可
```

如果执行结果为-10h：

源操作数	结果
8 位	DX=?　　，　AX=0FFF0h
16 位	DX=0FFFFh　　，　AX=0FFF0h

下面的实例示范了 16 位和 32 位的有符号数乘法：

```
        .DATA
mem16   SWORD   -3000
        .CODE
        ...
; 8-bit unsigned multiply
        mov   AL, 23        ; Load AL
        mov   BL, 24        ; Load BL
        mul   BL            ; Multiply BL, Product in DX: AX
                            ; overflow and carry set, 因为 23 * 24=552
                            ; 超过 AL 所能存储的大小，所以 AH 必不为零

; 16-bit signed multiply
        mov   AX, 50        ; Load AX                    50
                                                        -30000
        imul mem16          ; Multiply memory,          ------
                            ;   Product in DX: AX    -1500000
                            ; overflow and carry set, 因为-1500000
                            ; 超过 AX 所能存储的大小，所以 DX 必不为零
```

注意若结果的上半部（对 byte 指的是 AH，对 word 则是指 DX 或 EDX）是一个非零的

数，则会设置 OF 与 CF 为 1。

在 80186 到 80486 的处理器上，IMUL 指令还有另外三种额外的操作数结合方式。

(1) IMUL register16，immediate

register16 必是下列的寄存器之一：

AX　BX　CX　DX　SP　BP　SI　DI

实例如下：

```
MOV     BX，   10
IMUL    BX，   10   ；BX=BX * 10=100
```

此实例会将 BX 乘 10 的结果存储在 BX，这将改变原 BX 的值。

(2) IMUL register16，{memory16|register16}，immediate

实例如下：

```
IMUL  SI，   BX，   -5
IMUL  AX，  [BX]，6
IMUL  DX，  VAR1,5
```

这个操作数的范围同 (1)，也是目的操作数，执行的结果将存储于此。执行的情况是将第二与第三个操作数相乘，并将结果存储于第一个操作数。执行完，第二个操作数并不改变。

(3) IMUL　register，{register|memory}

实例如下：

```
IMUL  DX，   AX   ；Only on 80386/486
IMUL  AX，   [BX]
```

register（目的操作数）可以是 16 位或 32 位的寄存器，且源操作数必须和目的操作数有相同大小。(3) 只能使用在 80386/486 上。

(1) (2) 的运算结果总是存储在 16 位的寄存器中。若结果不超过 2 字节（16 位）则 CF=0，OF=0。若运算结果太大，则 CF=1，OF=1。(3) 可以有 16 位与 32 位的运算。对标志的影响同 (1) (2)。一般要使用 (1) (2) 的方法应先确定运算的结果不会超过 word 值。如果你不能确定，建议还是使用只有单一操作数的 MUL 与 IMUL 的格式。

7.4 除法运算指令

指令格式	O	D	I	T	S	Z	A	P	C
DIV　源操作数	?				?	?	?	?	?
IDIV　源操作数	?				?	?	?	?	?

DIV 及 IDIV 指令

DIV 指令执行无符号数（整数）的除法，IDIV 指令执行有符号数（整数）的除法。两个指令运算的结果都会得到一个商与一个余数。

源操作数可为寄存器或内存操作数。源操作数(除数)为 8 位时，被除数总是 AX。执行结果的商(quotient)存放在寄存器 AL 内，余数(remainder)存放在 AH 内。如果源操作数为 16 位，则被除数总是(DX:AX)，商存放在 AX 内，余数存放在 DX 内。如果源操作数为 32 位，则

被除数总是(EDX:EAX)，商存放在 EAX 内，余数存放在 EDX 内，如图 7-3 所示。

DIV 与 IDIV 指令实例如下：

(1) 8 位除数 (83h/2=41h，remainder=1)

```
MOV   AX,   83h   ; 被除数=AX
MOV   BL,   2     ; 除数=BL
DIV   BL          ; AL=41h (商),
                    AH=01h (余数)
```

被除数寄存器	除数大小	商	余数
AX	8 bits	AL	AH
DX：AX	16 bits	AX	DX
EDX：EAX	32 bits	EAX	EDX

图 7-3 除法运算

(2) 16 位除数 (8003h/100h=80h，remainder=3) 因为是 16 位的除法，所以 DX 为被除数的高 word，所以必须先清为零。除后余数存放在 DX 内，商存放在 AX 内。

```
MOV   DX,   0       ; 将被除数高 word 清为零
MOV   AX,   8003H   ; 被除数的低 word
MOV   CX,   100H    ; 除数=CX，被除数=DX：AX
DIV   CX            ; AX=80h (商)，DX=3h (余数)
```

(3) 同实例 (2) 我们以 16 位的内存变量当作除数：

```
MOV   DX,   WORD  PTR  DIVIDEND+2
MOV   AX,   WORD  PTR  DIVIDEND
IDIV  DIVISOR  ; AX=-5000，DX=-3
...
DIVIDEND  DWORD  -500003
DIVISOR   WORD    100
```

7.5 CBW、CWD、CDQ 和 CWDE 指令

指令格式

		O	D	I	T	S	Z	A	P	C
CBW	源操作数									
CWD	源操作数	±				±	±	±	±	±
CDQ	源操作数									
CWDE	源操作数									

CBW 指令可将在 AL 中的有符号数转换成一个 word 大小 AX 中，通过在 AL 中的最高位（即正负号位）延伸到 AH 中的所有位。如下：

```
(1) MOV   AL,   1   ; AH=?，AL=00000001B=01h
    CBW             ; AX=0001h
(2) MOV   AL，-1    ; AH=?，AL=11111111B—0FFh
    CBW             ; AX=0FFFFh
```

CWD 指令可将在 AX 中的有符号数转换成一个 dword 大小的寄存器组 DX：AX 中，通过在 AX 中的最高位（即正负号位）延伸到 DX 中的所有位。如下：

```
(1) MOV   AX，1     ; DX=?，AX=0001h
    CWD             ; DX：AX=0000：0001h
```

(2) MOV AX，−1 ；DX=?，AX=0FFFFh
CWD ；DX：AX=0FFFF：FFFFh

CWDE 指令（只对 80386/486）可将在 AX 中的有符号数转换成一个 dword 大小的 EAX 中，通过在 AX 中的最高位（即正负号位）延伸到 EAX 中的所有位。如下：

(1) MOV AX，1 ；AX=0001h
CWDE ；EAX=00000001h
(2) MOV AX，−1 ；AX=0FFFFh
CWDE ；EAX=0FFFFFFFFh

CDQ 指令(只对 80386/486)可将在 EAX 中的有符号数转换成一个 quadword 大小的寄存器组 EDX:EAX 中，通过在 EAX 中的最高位(即正负号位)延伸到 EDX 中的所有位。如下：

(1) MOV EAX，1 ；EDX=?，EAX=00000001h
CDQ ；EDX：EAX=00000000：00000001h
(2) MOV EAX，−1 ；EDX=?，EAX=0FFFFFFFFh
CDQ ；EDX：EAX=0FFFFFFFF：FFFFFFFFh

ADD 与 SUB 指令需要两个操作数大小相同，若要相加减两个大小不同的操作数，我们必须将较小的操作数值扩展成与较大的操作数相同的大小。传统的做法如下：

```
; unsigned numbers: add a byte to word
        MOV  AL, BYTE-VALUE
        MOV  AH, 0
        ADD  AX, WORD-VALUE
```

如果是对有符号数进行处理时，你必须小心去设置 AH 值以维持此数字正确的 2'S 补数表示。CBW（convert byte to word）指令可以帮你做这件事。如下：

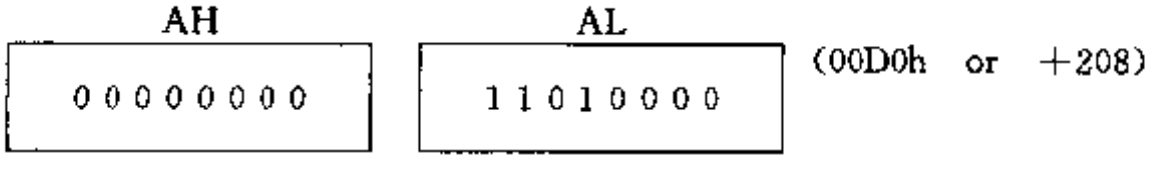
```
; signed numbers: add a byte to word
        MOV  AL, BYTE-VALUE
        CBW
        ADD  AX, WORD-VALUE
```

CBW 指令工作的范围只是 AH 与 AL。这个指令会检查在 AL 的值和设置 AH 的值，以得到与在 AL 相等的 2'S 补数值。如果 AL 是正数，AH 的所有位将设为 0；如果是负数，AH 所有的位将设为 1。

当我们执行有符号数的除法时，我们可能会误写成：

```
MOV   AH, 0
MOV   AL, −48   ; AX=00D0h
MOV   BL, 5
IDIV  BL
```

这样会得到错误的结果，因为 AL 的 sign bit 没有延伸到 AH，所以被除数为+208 而不是−48。



我们将上例改正如下，我们插入 CBW（convert byte to word）指令将位组（BYTE）的 sign bit 延伸到字组（WORD）。

	AH	AL
执行 CBW 前	?????????	11010000
执行 CBW 后	11111111	11010000

```
MOV   AL,   -48   ; AL=D0h
CBW               ; AX=0FFD0h
MOV   BL  ,  5
IDIV  BL          ; AL=-9 (0F7h),   AH=-3 (0Fdh)
```

如果我们使用 16 位的除数来看作有符号数的除法时，要记住使用 CWD（convert word to doubleword）将 sign bit 延伸到双字组（doubleword）。

```
MOV  AX, -5000  ; AX=0EC78h
CWD             ; DX: AX=0FFFFEC78h
MOV  BX,   256
IDIV  BX        ; AX=0FFEDh (-19d) 商
                ; DX=0FF78h (-136d) 余数
```

无符号数除法不需要注意标志的变化。下面的实例示范有符号数的除法，可能较复杂：

```
          .DATA
  mem16   SWORD   -2000
  mem32   SDWORD 50000
          .CODE
          :
          :
; Divide 16-bit unsigned by 8-bit
      mov   ax, 700                    ; Load dividend
      mov   bl, 36                     ; Load divisor
      div   bl                         ; Divide BL=36
                                       ; AL=19 (商)
                                       ; AH=16 (余数)
; Divide 32-bit signed by 16-bit
      mov   ax, WORD PTR mem32 [0]     ; Load into DX: AX
      mov   dx, WORD PTR mem32 [2]
      idiv mem16                       ; Divide memory=-2000
                                       ; AX=-250 (商)
                                       ; DX=0 (余数)
; Divide 16-bit signed by 16-bit
      mov   ax, WORD PTR mem16         ; Load into AX
      cwd                              ; Extend to DX: AX
      mov   bx, -421                   ; Divide by BX=-421
      idiv bx                          ; AX=4 (商)
                                       ; DX=-316 (余数)
```

Divide Overflow

当我们执行除法指令时，可能发生因商或余数太大而寄存器不能存放的情况，或是用户以 0 当除数时，DIV、IDIV 指令并不会设置溢位标志去表示一个溢位发生。CPU 本身也不作此种错误检查，当发生以上情况时，DOS 操作系统会调用中断 INT OH 去终止你的程序，并显示错误信息 Division overflow，并把控制权交回操作系统。在许多高级语言中，都有做 overflow 的检查，以防此错误的发生。

当我们遇到 32 位的被除数时，为避免 Divide overflow 的情况，我们可分成两个 16 位的操作数，个别去除。例如 01234567h/10h=00123456h（商），余数为 7。一看即知 AX 中放不下商，因为 AX 为 16 位，可容纳的最大值为 0FFFFh，明显小于 01234567h。如果硬用我们平常的方法去除，将产生 Divide Overflow，我们可利用下列实例的技巧，如下：

```
mov   AX, dividend+2    ; AX=0123h
cwd                     ; DX: AX=0000: 0123h (被除数)
mov   CX, divisor       ; CX=10h (除数)
div   CX                ; AX=0012h (商)
                        ; DX=0003h (余数)
mov   BX, AX            ; BX 存储商的高 word
mov   AX, dividend      ; DX: AX=0003: 4567h (商)
div   CX                ; CX=10h (余数), 没变
mov   remainder, DX     AX=3456H (商), DX=7h (余数)
; 所以 BX : AX=0012: 3456h (商), DX=7h (余数)
dividend   dword   01234567h
divisor    word    10h
remainder  word    ?
```

7.6 十进制数字

个人计算机存储数字有许多方法，一般是以二进制格式存储，现在我们要介绍以 10 为基址的十进制数字，有两种方式去做。如果每个十进制数字是存储在一个 byte 中，我们称此数字为非压缩的十进制数字。如果两个十进制数字存储在一个 byte 中，我们称此数字为压缩的十进制数字。

十进制数字有时被称为 Binary Coded Decimal 数字，通常缩写成 BCD。

在第 1 章已详细说明过 BCD 格式数字，此处不再多说。我们可以知道同样大小的数字，以 BCD 格式存储需要较多的内存。当一个数字要显示在屏幕时，需以字符的形式才可正确地输出。例如数字 12345，你必须先转换为字符 “1”、“2”、“3”、“4”、“5”。这个工作在高级语言中你不需自己去转换，在汇编语言中你必须自己做。处理器并没有直接执行十进制算术的指令，不过处理器提供特别的指令去调整二进制数字成十进制数字。

对于加、减、乘运算，你使用标准的 ADD、SUB、MUL 指令，然后再将结果转成十进制。对于除法，你必须先调整被除数，然后再执行除法。

加法和减法指令执行时，处理器会设置辅助进位标志 AF 去表示是否有调整的需要。这些

调整指令执行时会检查 AF，这是唯一使用此标志的情况。因此没有其它指令允许你直接去检查或修改 AF。

这些调整指令总是针对在寄存器 AL 内的值运算，因此当你使用十进制算术时，要记住把第一个操作数放在 AL 中。十进制算术有两个重要的限制：

(1) 你一次只能处理一个 byte。因此如果你使用非压缩十进制数字，你一次只能处理一位十进制数字。如果你使用压缩十进制数(1 byte 存储两个数字)，你一次能处理两位十进制数字。

(2) 压缩式十进制数字只可以被加和减，如果你要乘或除，必须先将数字转换成非压缩格式。

7.6.1 非压缩式 BCD 数字

当我们计算两个一位数字时产生 一个两位数字的结果时，AAA、AAS、AAM、AAD 指令会将运算结果的第一位数字（个位）放置在 AL 中，第二位数字（十位）放置在 AH 中。如果在 AL 的值需要从 AH 进位或借位时，这些指令会设置进位与辅助进位标志为 1。这四个对非压缩 BCD 的指令如图 7-4 所示。

指令	说　明
AAA	加法运算之后的调整
AAS	减法运算之后的调整
AAM	乘法运算之后的调整。总是使用 MUL，不使用 IMUL。
AAD	除法运算之后的调整。不像其它的 BCD 指令，AAD 是在运算前转换一个 BCD 的值为二进制值，在运算之后，使用 AAM 去调整商。注意，余数将会被覆盖掉。如果你需要余数，可以在调整商之前。先将它存储在另一个寄存器，稍后再移至 AL 中调整

图 7-4　指令说明

	O	D	I	T	S	Z	A	P	C
AAA	?				?	?	±	?	±
AAS	?				?	?	±	?	±
AAM	?				±	±	?	±	?
AAD	?				±	±	?	±	?

对于非压缩 BCD 数字的算术，你必须对每一位独立的数字做 8 位的算术计算，然后再将结果指定在寄存器 AL 中。在每个运算后，使用相关的 BCD 指令去调整结果。ASCII 调整指令并不会对操作数处理，它只能处理在寄存器 AL 内的值，所以指令之后不需指定操作数。

1. AAA 指令

AAA (ASCII adjust after addition) ASCII 加法调整指令。AAA 指令调整 ADD 或 ADC 计算后的二进制数结果。

AAA　；将二进制结果，调整成非压缩十进制数字。

这个指令会将寄存器 AL 的内容，调整成为非压缩 (unpacked) 十进制值。调整过程如下：

◆　如果寄存器 AL 的第 0～3 位所含的数值大于 9，或者当辅助进位标志 AF (Auxiliary Carry Flag) 被设为 1 时，CPU 会将 AL 加 6 并且将 AH 加 1，在执行完后 CF＝AF＝1。

◆　如果寄存器 AF 的第 0～3 位所含的数值小于或等于 9，或者当辅助进制标志 AF－0。令 CF＝AF＝0。

◆ 最后再将寄存器 AL 的 4～7 位清为零。

实例如下：

```
MOV   AX，36H   ；将 ASCII 数字'6'存放在寄存器 AL 中
MOV   BL，37H   ；将 ASCII 数字'7'存放在寄存器 BL 中
ADD   AL，BL    ；AL＝AL＋BL，得 AL＝6DH
AAA             ；AH＝01h，AL＝03h
```

下面 CH7-3. ASM 示范非压缩 BCD 格式数字（19，95）相加

CH7 _ 3. ASM

```
.DOSSEG
.286
.MODEL SMALL
.STACK 1024

    CLS         macro
                mov  ah, 7
                mov  al, 0
                mov  bh, 07h
                mov  cx, 0
                mov  dx, 184fh
                int  10h
endm
PUT-NUM  macro
            mov  ah, 2
            mov  dl, result [0]
            int  21h
            mov  dl, result [1]
            int  21h
            mov  dl, result [2]
            int  21h
endm

    .DATA
        message BYTE '  19', 0Dh, 0Ah
                     '+95', 0Dh, 0Ah,
                     '------', 0Dh, 0Ah,' $'
        num1    BYTE 1, 9
        num2    BYTE 9, 5
        result  BYTE 3 DUP (0)

.CODE
        .STARTUP
        mov ah, 9
```

```
mov dx, offset message
int 21h

    mov ah, 0
mov al, num1 [1]
add al, num2 [1]
aaa
add al, 30h
mov result [2], al
mov al, num1 [0]
add al, ah
mov ah, 0
add al, num2 [0]
aaa
add al, 30h
mov result [1], al

    add ah, 30h
mov result [0], ah
put _ num
.EXIT
END
```

num1、num2 变量各说明了两个元素，如同是有两个元素的数组。由于 1 或 5 或 9 只需 4 位存储，而又各存储在一个 byte 中，所以就像是非压缩 BCD 数字。两个数相加可能产生三位数，所以我们说明了占 3 bytes 的 result 变量。记住调整后的数字需加上 30h 才可得到正确的 ASCII 字符。

2. AAS 指令

AAS (ASCII adjuct after subtraction) ASCII 减法调整指令。AAS 指令调整 SUB 或 SBB 指令执行结果。

AAS　　；将二进制结果，调整成非压缩十进制数字。

这个指令会将寄存器 AL 的内容，调整成为非压缩（unpacked）十进制值。指令调整过程如下：

◆ 如果寄存器 AL 的第 0～3 位所含的数值大于 9，或者当辅助进位标志 AF (Auxiliary Carry Flag) 被设为 1 时，CPU 会将 AL 减 6 并且将 AH 减 1，在执行完后 CF＝AF＝1。

◆ 如果寄存器 AL 的第 0～3 位所含的数值小于或等于 9，或者当辅助进位标志 AF＝0，令 CF＝AF＝0。

◆ 最后再将寄存器 AL 的 4～7 位清为零。

实例如下：

```
MOV  AX, 103h     ; AX=0103h
MOV  BX, 4        ; BX=0004h
SUB  AL, bl       ; AX=00FFh, (F-6=9)
```

```
AAS                    ; AX=0009h
```

3. AAM 指令

AAM (ASCII adjust after multiplication) 乘法调整指令。AAM 指令可调整 MUL 指令的执行结果。

```
AAM     ; 将二进制结果，调整成非压缩十进制数字.
```

这个指令会将存放在寄存器 AL 内的两个非压缩十进制相乘的结果，调整成为非压缩 (unpacked) 十进制值。指令调整过程如下：

◆ 将寄存器 AL 的内容除以 10，商存放在 AH 中，余数存放在 AL 中。

实例如下：

```
MOV  AX, 0903h     ; AX=0903h
MUL  AH            AX=001Bh
AAM                AX=0207h，商=2，余数=7
```

4. AAD 指令

AAD (ASCII adjust before division) 除法调整指令。此指令先将非压缩的 BCD 码转成二进制码。

```
AAD     ; 调整 AX 内容成二进制数字，存放在 AL 中。
```

这个指令会将存放在寄存器 AX 内的非压缩十进制数字，调整成为二进制数字。它在执行除法指令之前使用。指令调整过程如下：

◆ 将寄存器 AH 的内容乘以 10，再加 AL 的内容，结果存放在 AL 中。

◆ 最后再将 AH 清为零。

实例如下：

```
mov  ax,  205h     ; AX=205H
mov  bl,  2        ; BL=2
AAD                ; AH=0, AL=19h
div  bl            ; AL=0Ch (商), AH=1 (余数)
AAM                ; AH=0Ch/10=1
                   ; AL=0Ch  MOD  10=2
```

原存储在 AH 的余数 1 将被破坏。可在执行 AAM 之前先存储起来。

7.6.2 压缩式 BCD 数字

压缩式 BCD 数字是由一个 byte 包含两个十进制数字组成；一个在高 4 位，一个在低 4 位。8086 家族处理器提供在加法与减法之后调整压缩 BCD 数字指令。对于乘法与除法就需你自己写子程序去调整。

对于压缩式 BCD 数字的算术，你必须对每一位独立的数字做 8 位的算术计算，然后再将结果存放在寄存器 AL 中。在每个运算后，使用相关的 BCD 指令调整结果。ASCII 调整指令并不会对操作数处理，它只能处理在寄存器 AL 内的值、所以指令之后不需操作数指定。

	O	D	I	T	S	Z	A	P	C
DAA	?				±	±	±	±	±
DAS	?				±	±	±	±	±

1. DAA 指令

DAA (Decimal adjust after addition) 加法调整指令。此指令调整 ADD 或 ADC 指令运算后存放在 AL 的结果。

DAA　　；将二进制结果，调整成压缩十进制数字。

这个指令会将寄存器 AL 的内容，调整成为压缩（packed）十进制值。调整过程如下：

◆ 如果寄存器 AL 的第 0～3 位所含的数值大于 9，或者当辅助进位标志 AF 被设为 1 时，CPU 会将 AL 加 6 并且令 AF＝1。

◆ 如果寄存器 AL 的第 4～7 位所含的数值大于 9，或者 CF＝1，则将寄存器 AL 加上 60h，且令 CF＝1。

2. DAS 指令

DAS（ASCII adjuct after subtraction）减法调整指令。DAS 指令调整 SUB 或 SBB 指令执行结果。

DAS　　；将二进制结果，调整成压缩十进制数字。

这个指令会将寄存器 AL 的内容调整成为压缩（packed）十进制值。指令调整过程如下：

◆ 如果寄存器 AL 的第 0～3 位所含的数值大于 9，或者当辅助进位标志 AF 被设为 1 时，CPU 会将 AL 减 6 并且令 AF＝1。

◆ 如果寄存器 AL 的第 4～7 位所含的数值大于 9，或者 CF＝1，则将寄存器 AL 减去 60h，且令 CF＝1。

实例如下：

```
; To add 88 and 33
    mov   ax, 8833h      ; AX=8833h
    add   al, ah         ; AH=88h,   AL=0BBh
    DAA                  ; AH=01h,   AL=21h,   CF=1
; To subtract 38 from 83 (=83-38)
    mov   ax, 3883h      ; AX=3883h
    sub   al, ah         ; AH=38h, AL=4Bh
    DAS                  ; AH=38h,   AL=45h,   CF=0
```

并不像其它的 ASCII 调整指令，十进制调整指令从不影响 AH。如果在低 4 位的数字对高 4 位的数字产生进位或借位，编译程序会设置 AF＝1。如果在高 4 位的数字需要从另一个 byte 进位或借位，编译程序会设置 CF＝1。

7.7 逻辑指令

CPU 执行某些基本的运算是以二进制的方式处理，布尔逻辑运算是其中最重要之一。它可以尽可能处理独立的位，如图 7-5 所示。

逻辑指令 AND、OR、NOT、TEST 和 XOR，可以改变和测试在一个 BYTE 或 WORD 内的位值。这些指令的格式如下：

指令	说　明
AND	当两位都是 1，结果是 1
OR	当有一位是 1，结果是 1
XOR	当一位是 1，另一位是 0，结果是 1
NOT	将 0 变 1，1 变 0

图 7-5　指令说明

			O	D	I	T	S	Z	A	P	C
AND	操作数 1	操作数 2	0				±	±	?	±	0
OR	操作数 1	操作数 2	0				±	±	?	±	0
XOR	操作数 1	操作数 2	0				±	±	?	±	0
NOT	操作数 1										
TEST	操作数 1	操作数 2	0				±	±	?	±	0

7.7.1 AND 指令

AND 指令执行布尔 AND 运算，操作数可以是 8 位或 16 位，且须是相同的大小，并将结果放在操作数 1。例如：

```
mov  ah, 01101010b
and  ah, 00001111b  ; AH=00001010b
```

操作数 1 是目的操作数，可以是寄存器或内存操作数。操作数 2 是源操作数，可以是寄存器、内存操作数或立即数。但是两个操作数只能有一个内存操作数。可以使用的实例如下：

```
and  ax, bx
and  al, bytevar
and  wordvar, dx
and  ah, 10h
and  bytevar, 00110010b
and  byte ptr [bx], cl
```

7.7.2 OR 指令

OR 指令执行布尔 OR 运算，操作数可以是 8 位或 16 位。且须是相同的大小，并将结果放在操作数 1。例如：

```
mov  ah, 01101010b
or   ah, 00001111b  ; AH=01101111b
```

操作数 1 是目的操作数，可以是寄存器或内存操作数。操作数 2 是源操作数，可以是寄存器、内存操作数或立即数，但是两个操作数只能有一个内存操作数，与 AND 指令相同，可以使用的实例如下：

```
or  cx, dx
or  cl, bytevar
or  al, 20h
or  wordvar, 0FFFFh
or  word ptr [bx], ax
```

利用 OR 指令可将单一的十进制数字转换为 ASCII 格式。例如 AL 包含一个数值 1h，为了输出，必须将它转换为 ASCII 码的 '1' =31h。只要将 AL 的第 4 和第 5 位设为 1，即加 30h 就可：

AL	0000	0001	(01h)
OR 运算	0011	0000	(30h)
结果	0011	0001	(31h) = '1' =49

汇编指令如下：

```
mov  al，01h
or   al，30h  ；AL=31h
```

还可用来测试数值是零或正负数，只需与自己做 OR 运算，再由标志值判断即可：

```
or   al，  al
```

如果 ZF=1，表示是零。如果 SF=1，则为负数。若都不是，必为正数。

7.7.3 XOR 指令

XOR 指令执行布尔 XOR 运算，操作数可以是 8 位或 16 位，且须是相同的大小，并将结果放在操作数 1。例如：

```
mov  ah，01101010b
xor  ah，00001111b  ；AH=01100101b
```

操作数 1 是目的操作数，可以是寄存器或内存操作数。操作数 2 是源操作数，可以是寄存器、内存操作数或立即数，但是两个操作数只能有一个内存操作数，可以使用的实例如下：

```
xor  ax，cx
xor  bx，wordvar
xor  dh，30h
xor  bytevar，7Fh
xor  byte ptr [bx]，30h
```

一般使用 XOR 指令的场合是在做屏幕切换时，它可恢复屏幕原来的属性。如下例：

```
mov  bh，00110011b  ；bh=00110011b
xor  bh，11111111b  ；bh=11001100b
xor  bh，11111111b  ；bh=00110011b
```

上例我们将存储在 BH 的值与 0FFh 做两次 XOR 正好可恢复原值。其实只要固定一个数，不一定要与 0FFh 或 0FFFFh 做 XOR 运算，与其它的数也可达到同样的结果。

7.7.4 NOT 指令

NOT 指令执行布尔 NOT 运算，可将操作数的所有位反转，也就是将 0 变 1，将 1 变 0。操作数可以是 8 位或 16 位，且须是相同的大小，并将结果放在操作数。例如：

```
mov  ah，00001111b
not  ah             ；AH=11110000b
```

操作数是目的操作数，可以是寄存器或内存操作数。可以使用的实例如下：

```
not  dh
not  cx
not  bytevar
not  wordvar
not  byte ptr [bx]
```

最能想到的应用就是取 2's 的补数，再利用 INC 指令就可取得此值的负数。如下：

```
mov  ah，1  ；AH=00000001b=01h (+1)
not  ah     ；AH=11111110b=0FEh (-2)
inc  ah       AH=11111111b=0FFh (-1)
```

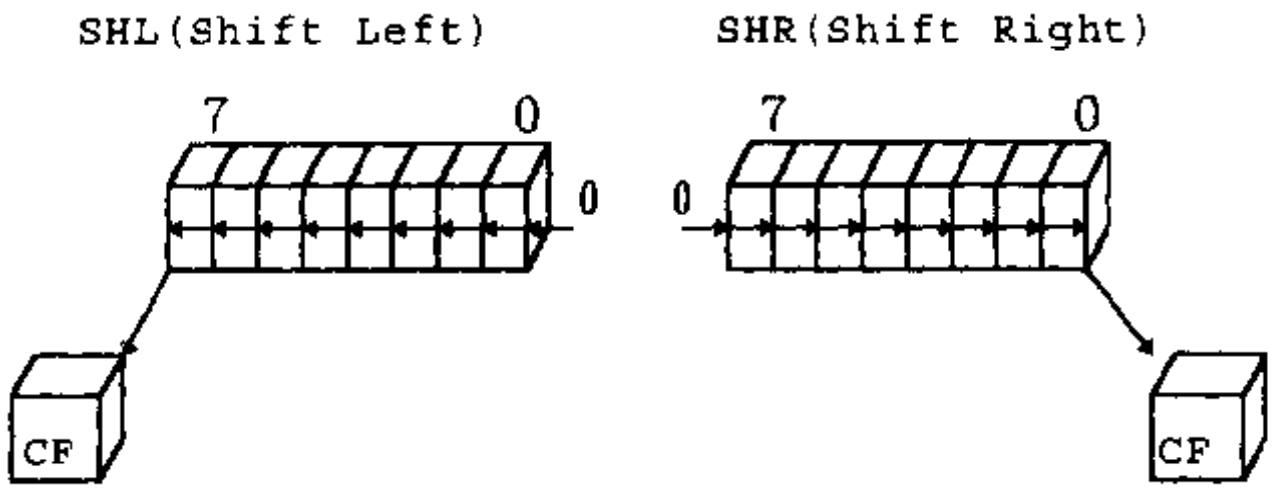


图 7-6　移位与循环移位

7.7.5　TEST 指令

TEST 指令也是执行布尔 AND 运算。操作数可以是 8 位或 16 位，且须是相同的大小，只是其结果只反应在标志值，而不改变任何操作数的值。当你想要知道操作数中某些特别的位时，非常有用。

操作数1是目的操作数，可以是寄存器或内存操作数。操作数2是源操作数，可以是寄存器、内存操作数或立即数，但是两个操作数只能有一个内存操作数，可以使用的实例如下：

```
test   ax, bx
test   al, bytevar
test   wordvar, dx
test   ah, 10h
test   bytevar, 00110010b
test   byte ptr [bx], cl
```

例如你想要知道位3和位7是否都是1，使用下面的方法。如果ZF=0，表示位0与3都是设为1（Yes）。

```
mov   al, 10001000b
mov   ah, 00001001b
test   al, 10001000b   ; AH, AL 不变, ZF=0, Yes
test   ah, 10001000b   ; AH, AL 不变, ZF=1, No
```

7.8 移位和循环移位指令

8086家族提供完整的移位和循环移位指令集。这些移位和循环移位指令提供在运算中移动位的方法，在一般高级语言中是很少的，但在汇编语言中是一个标准指令。

标准的指令有8个：SHL、SHR、SAL、SAR、ROL、ROR、RCL和RCR。指令格式如下：

		O	D	I	T	S	Z	A	P	C
SHL	目的操作数，移位次数	±				±	±	?	±	±
SHR	目的操作数，移位次数	±				±	±	?	±	±
SAL	目的操作数，移位次数	±				±	±	?	±	±
SAR	目的操作数，移位次数	±				±	±	?	±	±
ROL	目的操作数，移位次数	±								±
ROR	目的操作数，移位次数	±								±
RCL	目的操作数，移位次数	±								±
RCR	目的操作数，移位次数	±								±

7.8.1 SHL指令

SHL指令（图7-6）将目的操作数的每位往左移，最高（第7）位被移入CF，空出的最低（第0）位，以0填充。SHL指令有两种格式：

```
SHL   目的操作数, 1   ; 适用于 8088~80486
SHL   目的操作数, CL  ; 适用于 80186~80486
```

目的操作数可为8位或16位的寄存器或内存操作数。移动次数若为立即数1，可直接写在右边。若移动次数大于1，可将移动次数指定于寄存器CL。执行完后寄存器CL的值不改变。在80186~80486中，立即数可指定一个8位大小的整数常数。注意，若移动次数大于16，则结果将失去意义，因为目的操作数最多才16位。使用的实例如下：

```
(1) mov  al,   10000001b
    shl  al,   1            ; AL=00000010b, CF=1
(2) mov  wordvar,   00F0h
    mov  cl,   4
    shl  wordvar, cl        ; wordvar=0F00h, CF=0, CL=4
(3) mov  bl, 0Fh
    shl  bl, 4              ; BL=0F0h, 记得指定 .186 或以上
```

乘法与除法指令是相当慢的，特别是在 8088 和 8086 处理器上。当要乘 2 的次方时，移位指令可以加快运算的速度。因为往左移一位就等于乘 2，移两位就等于乘 2^2。依此类推。若不是 2 的次方，就不是很快了。下面的实例示范将 1 byte 大小的 bytevar 变量，利用 SHL 指令，得到 3 位的值，并将结果存储在 AL 中：

```
mov  bytevar, 1
mov  al, bytevar
shl  al,  1         ; AL=bytevar * 2
add  al,  bytevar   ; AL=bytevar * 3
```

由于 SHL 指令只能得到 2^N 次方的值，若要得到奇数倍的值，可利用上例的技巧。

7.8.2 SHR 指令

SHR 指令（图 7-6）将目的操作数的每位往右移，最低（第 0）位是被移入 CF，空出的最高（第 7）位，以 0 填充。SHR 指令有两种格式：

```
SHR  目的操作数, 1    ; 适用于 8088~80486
SHR  目的操作数, CL   ; 适用于 80186~80486
```

目的操作数可为 8 位或 16 位的寄存器或内存操作数。移动次数若为立即数 1，可直接写在右边。若移动次数大于 1，可将移动次数指定于寄存器 CL。执行完后寄存器 CL 的值不改变。在 80186~80486 中，立即数可指定一个 8 位大小的整数常数。使用的实例如下：

```
(1) mov  al,   10000001b
    shl  al,   1            ; AL=01000000b, CF=1
(2) mov  bl, 00000111b
    shr  bl, 4              ; BL=0h, CF=0 记得指定 .186 或以上
```

我们可以想像 SHR 指令可应用于除法。因为往右移一位就等于除 2，移两位就等于除 2^2。依此类推。对于一个有符号数的移位就需特别注意。最高位的正负号位，最好是用 SAR 指令，因为它可保留正负号。

下面的实例示范将 1 byte 大小的 bytevar 变量，利用 SHR 指令，得到除 8（2^3）的值：

```
mov  bytevar,  8
shl  bytevar,  3     ; bytevar=8/2^3=1,   .186 以上
```

7.8.3 SAL、SAR 指令

SAL 指令与 SHL 指令（图 7-6）的格式、执行的情况完全相同，只是我们将 SAL 指令，指定应用在有符号数的运算。其实两个往左移的指令并不会牵涉到正负号位，这就是为什么两个指令执行有相同的结果的原因。

SAR 指令与 SHR 指令的格式也相同。SAR 指令除了会像 SHR 指令一样将每位右移外，

右移的过程中，每次空出的最高位不再是以 0 填充，而会保留其正负号。也就是说，如果目的操作数是正数（即最高位是 0），还是与 SHR 一样补 0。若是负数（即最高位是 1），则每次右移一位都会将最高位补 1。由以下的实例你可清楚整个执行的过程与结果：

```
mov  ax, 8000h  ; AX=1000000000000000b=-32768
sar  ax,  1     ; AX=0100000000000000b=-16384, CF=0
sar  ax,  1     ; AX=0010000000000000b=-8192, CF=0
sar  ax,  1     ; AX=0001000000000000b=-4096, CF=0
sar  ax,  1     ; AX=0000100000000000b=-2048, CF=0
sar  ax,  1     ; AX=0000010000000000b=-1024, CF=0
sar  ax,  1     ; AX=0000001000000000b=-512, CF=0
```

7.8.4 ROL 指令

ROL 指令（图 7-6）会将目的操作数中的每位往左搬移。除了将最高位的值移入 CF 外，还会移入空出的最低位中，不会遗失此位值，这与移位指令是不同的。两种使用格式和移位指令相同：

```
ROL  目的操作数,  1   ; 适用于 8088~80486
ROL  目的操作数,  CL  ; 适用于 80186~80486
```

下面的实例可以很清楚地了解执行的过程与结果：

```
mov  al,  80h  ; AL=10000000b
rol  al,  1    ; AL=00000001b, CF=1
rol  al,  1    ; AL=00000010b, CF=0
rol  al,  1    ; AL=00000100b, CF=0
```

你可以使用 ROL 指令将操作数的上半部与下半部交换：

```
Exchang-byte-value:
mov  al, 12h    ; AL=12h
mov  cl, 4      ; 循环移位 4 次
rol  al, cl     ; AL=21h
Exchang-word-value:
mov  ax, 1234h  ; AL=1234h
mov  cl, 4      ; 循环移位 4 次
rol  ax, cl     ; AX=2341h
rol  ax, cl     ; AX=3412h
```

7.8.5 ROR 指令

ROR 指令（图 7-6）会将目的操作数中的每位往右搬移。除了将最低位的值移入 CF 外，还会移入空出的最高位中，不会遗失此位值，这与移位指令是不同的。两种使用格式和移位指令相同：

```
ROR  目的操作数, 1   ; 适用于 8088~80486
ROR  目的操作数, CL  ; 适用于 80186~80486
```

下面的实例可以很清楚地了解执行的过程与结果：

```
mov  al,  80h  ; AL=10000000b
```

```
ror    al,    1      ; AL=01000000b, CF=0
ror    al,    1      ; AL=00100000b, CF=0
ror    al,    1      ; AL=00010000b, CF=0
```

你同样可以使用 ROR 指令交换操作数的上半部与下半部。

7.8.6 RCL 指令

RCL 指令（图 7-6）会将目的操作数中的每位往左搬移。除了将最高位的值移入 CF 外，还会将原先存储在寄存器 CF 中的值移入空出的最低位中，这与 ROL 指令是不同的。两种使用格式和移位指令相同：

```
RCL   目的操作数，1   ；适用于 8088～80486
RCL   目的操作数，1   ；适用于 80186～80486
```

下面的实例使用了 CLC 指令清除进位标志。你可以很清楚地了解执行的过程与结果：

```
clc                  ; CF=0
mov   bl，88h        ; BL=10001000b
rcl   bl，1          ; BL=00010000b，CF=1
rcl   bl，1          ; BL=00100000b，CF=0
```

7.8.7 RCR 指令

RCR 指令（图 7-6）会将目的操作数中的每位往右搬移。除了将最低位的值移入 CF 外，还会将原先存储在寄存器 CF 中的值移入空出的最高位中，这与 RCL 指令是不同的。两种使用格式和移位指令相同：

```
RCR   目的操作数，1     ；适用于 8088～80486
RCR   目的操作数，CL    ；适用于 80186～80486
```

下面的实例使用了 STC 指令设置进位标志。你可以很清楚地了解执行的过程与结果：

```
stc              ; CF=1
mov   al，2      ; AL=00000010b，CF=1
rcl   al，1      ; AL=10000001b，CF=0
```

第8章　子　程　序

汇编语言的子程序与高级语言（如C、FORTRAN和Pascal）的functions、子程序，和procedure类似。有两个指令可以控制汇编语言的子程序。CALL会先将返回地址push至堆栈中，并转移控制权到子程序。RET会从堆栈中pop返回地址，并转移控制权到原程序中下个指令执行的地址。

我们可以将相关的子程序组织起来，供将来发展大程序时使用。尤其是一些经常性的工作，每次要使用时都重写一次岂不麻烦又费时费力。只要将它以程序库（library）的形式存储起来，下次要用时只要连接进来就可，这样就可节省很多时间。是不是觉得每次写个汇编小程序会花掉你相当多的时间，看到这里是不是有了那么一丁点兴趣，就让我们来领会一下本章的重要性。

8.1　子程序介绍

我们常在高级语言中看到子程序并使用它。在汇编语言里当然也有子程序（procedure）。将许多短而且有单一目的的程序集合在一起成为一个子程序，不仅可使一个大程序分成许多小程序，也可使编写及编译更容易及增加它的可读性，并使程序较结构化。

CPU在执行子程序之前，会利用堆栈区段先将主程序下一个执行的地址存储起来（也即是利用PUSH指令将地址推入堆栈而存储），再将被调用执行的子程序地址载入IP，以执行子程序，子程序最后会有个RET指令，将原先存储在堆栈的地址，再由POP返回IP，并将控制权交回原调用程序，以使程序能顺利正常执行下去。

例1：

```
.code
    main proc
    .
    .
main endp
sub1 proc
.
.
.
sub1 endp
sub2 proc
.
.
.
sub2 endp
```

例2：

```
.code
    main proc
    .
    .
    main endp
.code
    sub1 proc
    .
    .
    .
    sub1 endp
```

8.2 定义子程序

子程序在开头地方需要标记，在子程序结尾需有一个 RET 指令。正常子程序是在子程序开头使用 PROC 伪指令及结尾用 ENDP 伪指令所定义。RET 指令通常是紧放在 ENDP 伪指令之前。编译程序会确定 RET 指令的距离符合由 PROC 伪指令所定义的距离。

8.2.1 PROC 与 ENDP 伪指令

每个子程序的使用都需有一个 PROC 指令代表一个子程序的开头，而以 ENDP 代表一子程序的结束。PROC 可以自动保留某些不应该被更改，但在子程序执行可能改变的寄存器值。设立一个堆栈指针，以使你可以存取被放置在堆栈中的参数和区段变量。当子程序执行结束时。调整堆栈的内容。

基本文法如下：

```
label  PROC  [NEAR|FAR]
语句
       RET [常数]
label ENDP
```

8.2.2 RET 指令

每个子程序结尾（即最后一列）都会有一个 RET 指令，它的作用是告诉 CPU 此子程序执行结束。CPU 会将刚才存储在堆栈中原调用程序下一个指令地址再 POP 回 IP，以使程序能继续顺利执行下去，如果程序少写这一行程序就会无故当掉了，因为你也不晓得接下去，CPU 会执行到哪里。这是初学者常犯的错误，切记，切记！

CPU 的执行都是按照 IP 里存储的是什么指令就执行什么操作，所以我们可以利用这个特性来做控制权的转移，达到调用子程序执行完后还能回到原来程序继续执行。

8.2.3 CALL 指令

一般汇编语言程序又臭又长，我们鼓励将一个程序分割成许多有紧密组织、特定工作的小程序（子程序），如同高级语言一样，有结构化。由这些小程序来完成一个大程序的工作，不仅使可读性增加，如果有重复的工作，也只要写一次程序码，重复调用即可，可以节省程序设计的时间。

要调用子程序只要先将子程序写好。在程序中只要在 CALL 后面加上子程序名称就可达到调用的操作，还可重复使用。把它存储下来，下一次还可再用，如果子程序写多了，以后写程序只要写子程序就行了。

CALL 指令会自动将程序下一个执行的指令地址推（push）至堆栈，再将控制权转移至指定的地址执行。文法是：

CALL {label | register | memory}

操作数包含一个在执行时才能被计算的地址。操作数可以是寄存器、直接内存操作数、间接内存操作数。

调用（call）可以是近程（Near）或远程（Far）。近程调用只需 push 调用地址偏移部分，因此被调用的子程序（目的地）必须在相同的段或群组（group）中。你也可以指定目的操作数的类型。如果没有指定，MASM 会依照操作数（即标记，如我们常使用的 main）说明时的

距离（Near 或 Far），和适合的寄存器或内存操作数（变量）大小。编译程序会自动处理，就好像使用无条件转移的情况一样。

CALL 指令使用的语法如下：

CALL SUB

CALL NEAR PTR SUB1

CALL FAR PTR SUB2

NEAR PTR 表示原调用程序（如主程序）与被调用子程序是在同一个代码段（code segment）。当一个 NEAR 子程序被调用时，IP 值会加 1，以便稍候能指到下一个执行地址且 PUSH 到堆栈中。然后再把被调用子程序的偏移载入 IP，以执行子程序。因为在同一个代码段，CS 值固定，所以只要更改 IP 值（即将子程序的偏移载入 IP 执行即可），就可将控制权移转给子程序。最后碰到 RET 指令，CPU 会将刚才存储在堆栈中的地址，再 pop 回 IP（即将控制权交回给原调用程序），以便继续执行下去。

FAR PTR 表示被调用子程序，与原调用程序属于不同段（code），所以当 FAR 子程序被调用时，首先 CS 值会先 push 堆栈，然后将下一个执行地址再 push 堆栈。接着再将被调用子程序的代码段地址载入 CS，偏移载入 IP，以便执行子程序中的语句。执行完后是先 pop 回 IP 值，再 pop 回 CS，回到原调用程序继续执行。

※ 如果没有写 near ptr 或 far ptr 时，编译程序在编译时会假设为 near 属性，即假设在目前的代码段。

※ 虽然子程序有 near 与 far 之分，但 RET 指令使用仍相同。编译程序会分辨子程序为 near 或 far，而自动将 RET 指令改为 RETN（near）或 RETF（far），建议使用 RET，因为一定不会错。若自己改为 RETN 或 RETF 万一错了，反而麻烦！

CH8 _ 1. ASM

```
.dosseg
.286
.model small
.stack 100h
.code
    main proc
      . STARTUP
       call clrscr
       mov dx, offset message          ; 将字符串寻址载入 dx
       call print _ message
      .EXIT                            ; 程序结束回到 dos
main endp

clrscr proc                            ; 清屏幕且将光标移到左下角
       push ax
       push bx
       push cx
```

```
        push dx
          mov ax, 0600h              ; 令 al=0, 清除整个屏幕
          mov cx, 0
          mov dx, 184fh
          mov bh, 7
          int 10h
          mov ah, 2                  ; 将光标移至左上角
          mov bh, 0
          mov dx, 0
          int 10h
        pop dx
        pop cx
        pop bx
        pop ax
          ret                        ; return to main
    clrscr endp

    print _ message proc
        push ax
          mov ah, 9
          int 21h
        pop ax
          ret                        ; return to main
    print _ message endp

.data
    message byte 'This is first code segment data .', 0dh, OAH,'$'
end
```

上例程序 (CH8 _ 1. ASM) 在同一个代码段内使用三个子程序 (main、clrscr、print _ message)。main 调用 clrscr 子程序去清屏幕，调用 print _ message 子程序去打印出一个字符串。

clrscr 子程序利用 BIOS INT 10H 第六号功能将屏幕清除、第二号功能将光标移到左上角 (0, 0)。

print _ message 子程序利用 DOS INT 21H 第九号功能，将 DS: DX 地址为开头的字符串显示在屏幕上。

所有程序均使用同一个代码段，所以是使用 Call Near Ptr 方式调用，而 near ptr 可以省略不写，编译程序会假设为 Near。

一般我们在定义 procedure 或 label 时，都会定义一些较有意思的名称去代表一个子程序的名字，增加可读性。有时一个单字无法表达清楚子程序的功能，可能需两个或以上的单字汇编而成。在汇编语言子程序或标记的名称说明可以有底线“_”的存在，所以可以使用如 print _ message 所汇编的方式，定义一个较清楚的名称。

写子程序需要一个很好的习惯就是要保持子程序调用前后的寄存器值不变，以免破坏主程序的寄存器值，影响主程序的执行。利用 stack 在一进入子程序时就将会更动到的寄存器值 push 到 stack，而在子程序执行结束要写 ret 指令之前再 pop 回来。如果在子程序中需利用某个寄存器传送数据回到主程序时，就不要对这个寄存器做 push 和 pop 的操作，不然数据就传不回来。

8.3 Include 伪指令

include 是一个伪指令，用来把指定的文件含括（include）到程序中指定的位置。包含进来的文件会取代 include 指令而插入到 include 所在的位置。等于把包含的文件 copy 一份到指定的位置。使用 include 指令可省去重复写程序的麻烦，也便于程序的维护。

※如果包含的文件被修改过，所有使用它的程序，都必须重新编译并连接。

CH8 _ 2. ASM

```
.dosseg
.286
.model small
.stack 100h
.code
    main proc
        .STARTUP
            call clrscr
            mov dx, offset message
            call print _ message
        .EXIT
    main endp
    include clrscr.pro
    include display.pro
.data
    message BYTE 'This is first code segment data .', 0Dh, 0Ah,'$'
END
```

上例程序并没 clrscr 与 print _ message 这两个子程序的原始码，而是利用 include 指令，将这两个文件包含进来。

只要将 clrscr 子程序存在 clrscr.pro 文件中，print _ message 子程序存在 display.pro 文件中即可。你自己写文件名时当然可以自定。

clrscr.pro 文件内容如下

```
clrscr proc
  push ax
  push bx
  push cx
```

```
    push dx
    mov ax, 0600h
    mov cx, 0
    mov dx, 184fh
    mov bh, 07
    int 10h
    mov ah, 2
    mov bh, 0
    mov ch, 0
    int 10h
    pop dx
    pop cx
    pop bx
    pop ax
      ret
clrscr endp
```

display. pro 文件打印如下

```
print _ message proc
    push ax
      mov ah, 9
      int 21h
    pop ax
      ret
print _ message endp
```

注意此例 include 指令仍旧要写在 main 子程序之后、data 段之前，不要写错位置了。include 指令之后当然可以指定完整的路径，如 include c：\masm611\bin\clrscr. pro。

8.4 建立宏程序库

当然也可以将 clrscr、print _ message 这两个子程序改用宏，实例 CH8 _ 3. ASM 如下：

CH8 _ 3. ASM

```
. dosseg
. 286
. model small
. stack 100h
clrscr macro
      push ax
      push bx
```

```
    push cx
    push dx
      mov ax, 0600h
      mov cx, 0
      mov dx, 184fh
      mov bh, 07
      int 10h
      mov ah, 2
      mov bh, 0
      mov dx, 0
      int 10h
    pop dx
    pop cx
      pop cx
      pop bx
      pop ax
    endm clrscr
print _ message macro
    push ax
       mov ah, 9
       int 21h
    pop ax
endm print _ message
.code
    main proc
       .STARTUP
       clrscr
       mov dx, offset message
       print _ message
       .EXIT
    main endp
.data
       message BYTE 'This is first code segment data.', 0Dh, 0Ah,'$'
end
```

通常我们会将较常使用的宏放在同一个文件中，再利用 include 包含进来。实例 CH8 _ 4. ASM 如下：

CH8 _ 4. ASM

```
.dosseg
.286
.model small
.stack  100h
```

```
include macro.h
.code
    main proc
        .STARTUP
         clrscr
         mov dx, offset message
         print _message
        .EXIT
    main endp
.data
      message byte 'This is first code segment data.', 0DH, 0AH,'$'
end
```

macro.h 文件的内容如下：

MACRO.H

```
clrscr macro
    push ax
    push bx          ;  可改用 PUSHA 指令
    push cx
    push dx
    mov  ax , 0600h
    mov  cx , 0
    mov  dx , 184fh
    mov  bh , 07
    int  10h
    mov  ah , 2
    mov  bh , 0
    mov  dx , 0
    int  10h
    pop dx
    pop cx           ;  可改用 POPA 指令
    pop bx
    pop ax
  endm clrscr
print _message macro
    push ax
      mov ah , 9
      int 21h
    pop ax
  endm print _message
```

macro.h 文件内有两个宏，由此我们知道可以将平常较常用到的宏都存到同一文件内，就如同一个宏程序库一样，下次还可再使用，又可以少写好几行了。macro.h 文件名当然可以由用户自定。子程序文件名用 .h，使用 C 的用户，更能体会它的用意。子程序文件名用 .h 当然不是规定的，你喜欢写什么就写什么，不过最多只能有三个字。

注意 macro.h 要写在 code 之前，可不要写在 main endp 和 .data 之间，且宏内不要写 RET。有没有注意到这和子程序不同，宏要写在调用它之前，而子程序倒可以写在使用它之后，因为 call 指令会帮我们找到它并将控制权交给它。

宏和子程序的差别

宏程序在执行时，并没有像子程序一样，要做控制权转移，因为编译程序会将程序中宏名称出现的地方，将宏内所有指令插入到宏名称出现的位置。宏程序可任意传递参数，每次传递参数不同，执行的结果也会不同。而子程序不能传递参数，不过还是可以通过寄存器将欲传的参数数据先存放在某个寄存器，而子程序就固定地使某个寄存器的数据当做参数使用。所以在调用子程序执行时要特别注意在子程序执行之前寄存器的内容。例如 print _ message 子程序就利用 DS:DX 地址当做传递参数的方法，所以 print _ message 子程序在使用之前，要特别指定 DX 的内容，不然执行结果就不是我们所能想像的了。

新版的 MASM 已可利用某些方法来传递参数，不过多用在与高级语言连接时使用，这并不包括在本书范围之内。

宏程序的执行速度比子程序快，因为 call 及 ret 指令占用了执行的时间。宏程序的执行较占内存，因为每执行一次其所代表的指令会在实际的内存中占用一份空间。而子程序在执行完后，就会将所占用的内存释放掉，而下一个子程序就可在相同的空间上执行而不占较多内存。

8.5 EXTERN 和 PUBLIC 伪指令

在汇编语言或高级语言中，任意一个程序都常会用到一些 keyboard 进行输入，屏幕进行输出的处理。而在汇编语言中，你可能需要对每一细微的操作自己做处理。而编写一些又繁琐又长的程序是很繁复的事情。现在我们找到一个方法就是使用程序库（library）。

在前面我们学会了利用 include 指令来缩减程序编写的时间，但若使用过多的 include 文件，将会减缓编译的速度。因为 include 文件中有一些可能是我们不须用到的，且程序若太大，编译程序可能无法处理。

最好是将程序分成许多模块（module），每一个模块分开编译成一个 obj 文件，再利用 link 程序将需用到的模块连接在一起成为一个 EXE 文件。

一个结构化程序设计的原则就是模块化（modularity），每一模块之间数据的传递需要一个界面（interface），使得程序数据或参数的传递能正常而不致混淆。

进行两个程序连接，EXTERN 和 PUBLIC 两个伪指令是一定要用到的。

8.5.1 EXTERN 伪指令

这和 C 的 EXTERN 指令类似，EXTERN 使用的语法格式如下：

EXTERN　name：type

ENTERN 会指定编译程序，name 在外部程序中，name 的地址在 link 时再填入它的地址。即 name 是说明在其它的程序中，在编译时先将它的地址空着，待 link 时再填入。

※ 5.0 版之前是使用 EXTRN，现已改为 EXTERN。

name 可能是一个子程序或标记的名称，而 type 可能是一个大小或是有关这个 name 的某个属性。可能使用 type 的情况列在表 8-1 中。

表 8-1

type	意 义
ABS	EQU 运算符说明
PROC	缺省值为一子程序
NEAR	同段的子程序
FAR	不同段的子程序
BYTE	1 byte 大小
WORD	2 bytes 大小
DWORD	4 bytes 大小
FWORD	6 bytes 大小
QWORD	8 bytes 大小
TBYTE	10 bytes 大小

EXTERN clrscr：PROC 表示要存取位于另一文件中的子程序（外部子程序）

※ 程序若都是使用简化段指令（如 .code.data.model... 等），当 extern 指令后面所接 type 是 proc 时，均表示是一个子程序。如果程序是说明使用 .model small 模式，则所有 proc 的 type 均假设为 near。若为 medium、large 或 huge 模式，则 proc 的 type 均假设为 far。

EXTERN clrscr：NEAR 在 small 模式下的程序，使用外部子程序应说明为 near。

EXTERN clrscr：FAR 在 medium、huge、large 模式下的程序，使用外部子程序应说明为 far。

※ 若程序是使用简化段指令的话，直接写 proc 即可，不必自己指定 near 或 far 了，编译程序会自动帮我们决定为近程或远程调用。

EXTERN true：ABS，false：ABS

表示所要参考的外部变量是用 EQU 运算符说明。若有两个或两个以上的变量，它们之间可用逗号隔开，或各自写一行，如下：

EXTERN true ：ABS

EXTERN false：ABS

EXTERN one-byte：BYTE，twobyte：WORD，tenbyte：TBYTE

表示所要参考的外部变量是以 byte、word、tbyte・・・・说明。

8.5.2 PUBLIC 伪指令

当一个程序模块要能被其它程序模块所存取使用，就可以利用 PUBLIC 指令。一般格式如下：

PUBLIC NAME [，NAME，・・・・]

PUBLIC 指令只要写在 NAME 说明之前即可，如下：

```
.dosseg
.model small
   public num      <-------①
 num equ 10
 message equ 'dos error'
            <-------②
.code
```

```
public clrscr ←————③
 clrscr proc
    :
 clrscr endp
```

• public num 一定要写在①区。

• public clrscr 则写在②或③区均可。

※ name 若是常数名(即用 EQU 运算符说明),则只能表示 1 或 2 byte 大小,不能包含字符串常数。像上例的 message 就不能说明为 PUBLIC。

现在将 clrscr 子程序存在 clrscr. asm 文件中,print _ message 子程序存在 display. asm 文件中,主程序存在一个名为 test1. asm 文件中。下面为 test1. asm 文件的内容。

```
.dosseg
.286
.model small
.stack 100h
      extern clrscr : proc, print _ message : proc
.code
      main proc
           . STARTUP
            call clrscr
            mov dx, offset message
            call print _ message
           . EXIT
      main endp
.data
      message byte 'This is first code segment data.', 0DH, 0AH, '$'

end
```

下面为 clrscr. asm 文件的内容。

```
.model small
        public clrscr
.code
    clrscr proc
          push ax
          push bx
          push cx
          push dx
            mov ax , 0600h      ; aL=0, clear screen
```

```
    mov cx, 0
    mov dx, 184fh
    mov bh, 07
    int 10h
    mov ah, 2          ; set cursor (0, 0)
    mov bh, 0
    mov dx, 0
    int 10h
  pop dx
  pop cx
  pop bx
  pop ax
    ret
clrscr endp
end ————→; 不要忘了
```

下面为 display. asm 文件的内容。

```
. model small
        public print _ message
. code
    print _ message proc
        push ax
          mov ah , 9
          int 21h
          pop ax
        ret
    print _ message endp
end
```

test1. asm 为我们的主程序。程序中说明 clrscr 及 print _ message 两个子程序为外部子程序，在程序中直接调用它们。而在 clrscr. asm 中存在 clrscr 的子程序，且说明可为其它程序引用(public)，display. asm 中存在 print _ message 的子程序。在这两个子程序中使用 . MODEL，. CODE 指令是为了维持与主程序的兼容性。而在结尾 end 指令后无任何字（标记），是因为一个程序执行只能有一个进入点，就是在主程序中的 main proc，而子程序 END 指令无需指明结束（end）哪一个进入点，就可以由任一程序引用。

准备操作有两个步骤：

（1）将 test1. asm、display. asm、clrscr. asm、ML. EXE，放在同一个工作目录下，然后在 DOS 命令行下，执行

（2）ML test1，clrscr，display

执行完后将产生

TEST1.ASM，TEST1.OBJ，TEST1.EXE

CLRSCR.ASM，CLRSCR.OBJ

DISPLAY.ASM，DISPLAY.OBJ

整个编译连接的工作一次完成。执行文件在 test1.ext。正常情况下，我们会将主程序（test1.asm）写在最前面，因为最后的执行文件将放在最前一个文件中。你可以试试将 clrscr.asm 或 display.asm 写在最前面照样产生 clrscr.exe 或 display.exe，一样可以执行，是不是很有趣。建议将主程序写在最前面，因为这样比较符合一般的习惯。

注意 clrscr.asm 和 display.asm 的顺序是可以颠倒的。

Extern 大都是在所有段开头位置说明的，即 .code 的下一列或上一列，只有 FAR 属性或 ABS 属性的常数可以在程序中任何位置说明。如果都在段开头说明则较有规则也较易维护、查看。

Public 之后的 name 是在编译阶段处理的。link 并不处理 name 数据，link 无法把不同模块的 name 连接起来，因此不同模块的程序就不能相互调用。同样地，不同模块的数据也不能相互存取。不过如果 name 说明为 public，则编译程序便会把 name 的数据也写入 .OBJ 文件中，此时 link 便可依此数据来连接不同模块的数据，而使不同模块的程序能相互调用、相互存取数据。

检查一下 test1.ASM 文件中使用了三个段（stack、code、data），clrscr.ASM 使用了一个段（code），display.ASM 也使用了一个段（code），可是连接后观察 MAP 文件只有三个段。新版的编译程序会将相同名称的段 link 成一个段。而使各模块能兼容。

8.5.3 参数传递

谈到子程序就需谈到参数的传递。通过不同参数的传递可使子程序的执行每次都有不同的执行结果。

一般传递参数有两个方法：①通过寄存器或变量。②利用堆栈。

通过寄存器或变量的传递是直接也是最方便的方法，当然速度也最快。在调用子程序之前，先在特定寄存器中存放要传递的参数，子程序每次都会去存取特定寄存器的值去执行。缺点是，它将减少其余可用寄存器的总数，因为寄存器总数是有限的。

利用堆栈的方式可以节省寄存器的使用，且能传递较多量的数据。缺点是，这将减缓程序执行的速度，因为必须多做一些 push 及 pop 的操作。这也是一般高级语言使用的方法。

在汇编语言中，子程序参数传递的方法，两者皆可。视个人的程序设计风格及各种情况下的考虑。

8.6 LIBRARY

到目前已写过许多 I/O 方面的程序，而且几乎在每个程序中都会用到其中某几个。如果每天都重写一次那不是累死。我们准备将以前所学的许多有用的程序结合在一起成为一个程序库。前面我们使用 LINK 程序每次都需在 LINK 之后写一大串，那也不是一个好方法。现在有一个好消息要说明，就是利用 LIB 程序将所有 OBJ 文件都连接到一个 LIBRARY 文件中，以后程序只要指定一个 LIB 文件即可，那不是更省事了吗！以下是我们准备连接在一起的子程序列表（表 8-2）。

表 8-2　子程序列表

子 程 序	功 能 描 述	存 放 的 文 件
setvmode	设置显示方式	setvmode. ASM
cursrlen	设置光标大小	cursrlen. ASM
gotoxy	设置光标位置	gotoxy. ASM
getxy	读取光标位置	getxy. ASM
setvpage	设置有效显示页	setvpage. ASM
scrolwin	将屏幕往上卷或往下卷	scrolwin. ASM
clrscr	清除屏幕数据	clrscr. ASM
getvmode	取当前显示方式	getvmode. ASM
getche	输入一字符且回应至屏幕	getche. ASM
getch	输入一字符且没有回送至屏幕	getch. ASM
putchar	输出一字符至屏幕	putchar. ASM
readkey	输入一字符且不等待不回应至屏幕	readkey. ASM
getstr	输入一字符串	getstr. ASM
putstr	输出一字符串	putstr. ASM
keystat	检查键盘状态	setvmode. ASM
clearkey	清除键盘缓冲区后等待输入	setvmode. ASM
writline	光标跳到下一列开头	setvmode. ASM

8.6.1　独立子程序的描述及程序列表

1. setvmode：设置显示模式。显示模式数值指定在 Video _ Mode。

(1) 使用实例

```
mov   video _ mode，3 ；为 80 * 25 彩色模式
call    setvmode
```

(2) 程序列表 (setvmode. ASM)：

```
. model small
    public setvmode，video _ mode
. data
    video _ mode BYTE ?
. code
  setvmode proc
    push ax
      mov ah，0
      mov al，video _ mode
      int 10h
    pop ax
      ret
  setvmode endp
```

```
        end
```

2. cursrlen：设置光标大小。光标起始线在 Cursor _ Top，终止线在 Cursor _ bottom。

(1) 使用实例：

```
    mov   cursor _ top, 13 ; VGA 卡光标大小
    mov   cursor _ bottom, 14
    call    cursrlen
```

(2) 程序列表 (cursrlen. ASM)

```
        .model small
                public cursrlen, cursor _ top, cursor _ bottom
        .data
                cursor _ top      BYTE ?
                cursor _ bottom BYTE ?
        .code
            cursrlen proc
                push ax
                push cx
                    mov ah, 1
                    mov ch, cursor _ top
                    mov cl, cursor _ bottom
                    int 10h
                pop cx
                pop ax
                    ret
            cursrlen endp
        end
```

3. gotoxy：设置光标位置。显示页数在 Show _ Page，列数在 Row，行数在 Column。

(1) 使用实例：

```
    mov   show _ page, 0   ; 第 0 页
    mov   row,    0        ; 列数=0
    mov   column,    0     ; 行数=0
    call    gotoxy
```

(2) 程序列表 (gotoxy. ASM)

```
        .model small
                extern show _ page: BYTE
                public gotoxy, row, column
        .data
                row         BYTE ?
                column    BYTE ?
        .code

            gotoxy proc
                push ax
```

```
        push bx
        push dx
            mov ah, 2
            mov bh, show _ page
            mov dh, row
            mov dl, column
            int 10h
        pop dx
        pop bx
        pop ax
            ret
    gotoxy endp
end
```

4. getxy：读取光标位置。读取的显示页数在 Show _ Page，传回光标位置在 Row 与 Column 上，光标扫描线在 Cursor _ Top 与 Cursor _ Bottom 上。

(1) 使用实例：

```
mov   show _ page, 0   ; 设置读取第 0 页
call   getxy
mov   row, dh
mov   column, dl
mov   cursor _ top, ch
mov   cursor _ bottom, cl
```

(2) 程序列表 (getxy. ASM)

```
.model small
        extern show _ page: BYTE
        public getxy
.code
    getxy proc
      push ax
      push bx
        mov ah, 3
        mov bh, show _ page
        int 10h
      pop bx
      pop ax
        ret
    getxy endp
end
```

5. setvpage：设置有效显示页数。设置的显示页在 Show _ Page 上。

(1) 使用实例：

```
mov   show _ page, 0   ; 第 0 页
```

```
call    setvpage
```

(2) 程序列表 (setvpage. ASM)

```
.model small
        public setvpage, show _ page
.data
      show _ page BYTE?
.code
      setvpage proc
          push ax
            mov ah, 5
            mov al, show _ page
            int 10h
            pop ax
              ret
        setvpage endp
    end
```

6. scrolwin：将屏幕往上卷或往下卷。Scroll _ Function _ Number＝06H 或 07H。卷动列数在 Scroll _ Line _ Number。左上角位置在 Up _ Left _ Corner，右下角位置在 Dow _ Right _ Corner。屏幕属性在 Video _ Attribute。

(1) 使用实例：

```
mov   scroll _ function _ number, 06
mov   scroll _ line _ number, 25
mov   video _ attribute, 07h
mov   up _ left _ corner, 0
mov   down _ right _ corner, 184Fh
call    scrolwin
```

(2) 程序列表 (scrolwin. ASM)

```
.model small
        public scrolwin, scroll _ function _ number,
        scroll _ line _ number, video _ attribute,
        up _ left _ corner, down _ right _ corner
.data
      scroll _ function _ number BYTE?
      scroll _ line _ number     BYTE?
      video _ attribute          BYTE?
      up _ left _ corner         WORD?
      down _ right _ corner      WORD?
.code
      scrolwin proc
          push ax
          push bx
```

```
push cx
push dx
  mov ah, scroll _function _number
  mov al, scroll _line _number
  mov bh, video _attribute
  mov cx, up _left _corner
  mov dx, down _right _corner
  int 10h
pop dx
pop cx
pop bx
pop ax
  ret
      scrolwin endp
end
```

7. clrscr：清除屏幕数据。利用卷动屏幕功能并令 AL＝0，达到清屏幕效果，并将光标移动到屏幕左上角第一个位置。

（1）使用实例：

```
call    clrscr
```

（2）程序列表（clrscr. ASM）

```
. model small
        public clrscr
. code
      clrscr proc
            push ax
            push bx
            push cx
            push dx
              mov ax, 0600h      ; aL＝0, clear screen
              mov cx, 0
              mov dx, 184fh
              mov bh, 07
              int 10h
              mov ah, 2          ; set cursor (0, 0)
              mov bh, 0
              mov dx, 0
              int 10h
pop dx
pop cx
pop bx
pop ax
  ret
```

```
        clrscr endp
    end
```

8. getvmode：取当前显示方式。传回值 AH＝屏幕宽度（行数），AL＝显示模式，BH＝显示页数。

（1）使用实例：

```
call    getvmode
mov   column，ah
mov   video _ mode，al
mov   show _ page，bh
```

（2）程序列表（getvmode. ASM）

```
    . model small
        public getvmode
    . code
        getvmode proc
            mov ah，OFh
            int 10h
            ret
        getvmode endp
    end
```

9. getche：会等待输入一字符，且输入的字符会显示在屏幕上。传回的输入字符在 AL。

（1）使用实例：

```
call    getche
mov   char，    al
```

（2）程序列表（getche. ASM）

```
    . model small
        public getche
    . code
        getche proc      ; return char in aL
            mov ah，1
            int 21h
            ret
        getche endp
    end
```

10. getch：会等待输入一个字符，但不回应于屏幕。

（1）使用实例：

```
call    getch
mov   char，al
```

（2）程序列表（getch. ASM）

```
    . model small
        public getch
    . code
```

```
getch proc      ; return char in aL.
      mov ah, 8
      int 21h
      ret
getch endp
end
```

11. putchar：输出一字符至屏幕。输出的字符指定在 Char。

(1) 使用实例：

```
mov  char, 65      ; A′=65
call   putchar
```

(2) 程序列表 (putchar. ASM)

```
. model small
      public putchar, char
. data
      char BYTE?
. code
     putchar proc      ; pass char in dL
       push ax
       push dx
         mov ah, 2
         mov dl, char
         int 21h
       pop dx
       pop ax
         ret
     putchar endp
end
```

12. readkey：输入一个字符，不回应至屏幕，也不等待。若没有按键，则 ZF＝1，若有按键，则输入的字符传回在 AL，ZF＝0。

(1) 使用实例：

```
call  readkey
jz  no _ input
mov  char, al
:
:
no _ input:
```

(2) 程序列表 (readkey. ASM)

```
. model small
      public readkey
. code
     readkey proc      ; return char in aL
```

```
        push dx
          mov ah, 6
          mov dl, OFFh
          int 21h
        pop dx
          ret
    readkey endp
end
```

13. getstr：输入一字符串。DS：DX 为存放输入字符串的地址。

(1) 使用实例：

```
mov   dx, offset input _ buffer
call    getstr
```

(2) 程序列表 (getstr. ASM)

```
. model small
        public getstr
. code
      getstr proc      ; pass input buffer offset in dx
        push ax
          mov ah, OAh
          int 21h
        pop ax
          ret
      getstr endp
end
```

14. putstr：输出一字符串。DS：DX 为存放输出字符串的地址。

(1) 使用实例：

```
mov   dx, offset output _ buffer
call    putstr
```

(2) 程序列表 (putstr. ASM)

```
. model small
        public putstr
. code
      putstr proc      ; pass string offset in dx
        push ax
          mov ah, 9
          int 21h
        pop ax
          ret
      putstr endp
end
```

15. keystate：检查键盘状态。传回值 AL=OFFH 表示有字符未读出，AL=0 表示没有字符。

(1) 使用实例：

```
call    keystate
```

(2) 程序列表 (keystat.ASM)

```
.model small
        public keystat
.code
    keystat proc        ; return key buffer status in aL
        mov ah, OBh
        int 21h
        ret
    keystat endp
end
```

16. clearkey：清除 Key Buffer，等待输入。输入的功能号码放在 AL。

(1) 使用实例：

```
mov   al, 1
call    clearkey
```

(2) 程序列表 (clearkey.ASM)

```
.model small
        public clearkey
.code
    clearkey proc           ; pass input option in aL
        push ax             ; aL may be 01h, 06h, 07h, 08h or other
          mov ah, 0Ch
          int 21h
        pop ax
          ret
    clearkey endp
end
```

17. writline：跳到下一列开头位置。

(1) 使用实例：

```
call writline
```

(2) 程序列表 (writline.ASM)

```
.model small
        extern putchar: proc, char: byte
        public writline
.code
    writline proc
        push ax
        push dx
            mov ah, 2
            mov char, 0Dh
            call putchar
```

```
            mov char，0Ah
            call putchar
        pop dx
        pop ax
          ret
    writline endp
end
```

使用 LIB，你可以建立一个 LIBRARY 文件，增加模块到程序库，删除或取代模块。你可以结合程序库到一个程序库文件和拷贝或搬移一个模块到一个独立的目的文件。你也可以产生在程序模块中所有公用符号的列表。

在 DOS 命令行中执行 LIB 有两种方式：

①你可以将在命令行下所有需要的输入一次指定在同一列命令行中。格式如下：

LIB oldlibrary [option] [command] [，listfile] [，newlibrary] [;]

各区段必须按顺序出现，且会忽略所有的空白。而“;”可以使用在任何区段之后终止命令；LIB 会将其余的区段采用缺省值。

※ 可以在执行的过程中按 Ctrl+C 终止执行。

下列实例指示 LIB 将两个目的模块 FIRST.OBJ 和 SECOND.OBJ 结合到一个名为 THIRD.LIB 的程序库中：

LIB FIRST+SECOND,, THIRD

②如果你没有在命令行指定所有需要的输入和在最后以“;”结束，LIB 会针对所缺少输入的区段打印提示信息，等待你输入，直到所有输入都完成。如下：

Library name; oldlibrary [option]

Operations; commands

List file; listfile

Output library; newlibrary

你可以在任何时候输入“;”立即按 Enter 结束其余输入。当然剩余的区段仍采用缺省值。

8.6.2 区段说明

本章节描述 LIB 需要的输入区段。这些区段有 oldlibrary、options、commands、listfile 和 newlibrary。

1. 程序库文件（library）

oldlibrary 区段指定一个存在的或将要建立的程序库名称。如果你省略了子程序文件名，LIB 会假设子程序文件名为 .LIB。当然你可以指定一个完整的路径和文件名。

此路径和文件名不可包含“-”字符，LIB 会将此字符解释为 LIB 的“删除”运算符。

（1）建立一个程序库文件

要建立一个新的程序库文件，必须在命令行的 oldlibrary 区段或在提示字符串 library name; 之后指定文件名称。LIB 会假设子程序文件名为：LIB。

这个新的程序库文件名不可以是已存在的文件。如果是，LIB 会假设你是要改变此存在文件的内容。当你给定的文件名确定是不存在时，LIB 会显示下列的提示字符串：

Library file does not exist. Create?

此时按“Y”键表示要建立此文件，按“N”键则终止整个过程（工作）。如果在文件名之后立即接上命令（commands）、“,”或“;”，则 LIB 会认为你是确定要建立此程序库，而不会打印提示字符串，也就是假设为“Y”。

（2）执行检查

如果在 oldlibrary 区段之后立即加一个“;”，LIB 会检查在此程序库中所有的模块是否是可用的型式。如果发现一个无效的目的模块，则会打印出信息。如果所有模块都是完整的，则不会有任何信息出现。

下列的实例造成 LIB 去执行对 FOR. LIB 的检查（如果此文件存在）：

LIB FOR;

2. LIB 选项（Options）

所有选项是不分大小写的，只可以出现在 oldlibrary 和 commands 区段之间或 Library Name：之后。所有选项之前都需有一个“/”（选项指定字）。千万不要使用“/”当选项指定字，因为 LIB 会解释为“删除”运算符。选项也可以使用缩写名称，即下列各选项中括号内的名称可省略不写：

（1）/H［ELP］

调用 QuickHelp 公用程序。如果 LIB 找不到 QuickHelp 的辅助说明文件（Help file），将显示 LIB 命令行语法的简略摘要。

（2）/I［GNORECASE］

告诉 LIB，当比较符号名称时，不要比较大小写（缺省值）。使用/NOI 选项去建立程序库时会分大小写。当将分大小写的程序库与不分大小写的程序库结合时，可使用/IGN 去建立一个不分大小写的程序库。

（3）/NOE［XTDICTIONARY］

告诉 LIB，不要在模块间建立一个交叉参考的延伸表格。LINK 会使用它去加快程序库的搜寻。而/NOE 的意思就是不要去读它（“dot not read an extended dictionary.”）。

建立一个延伸表格需要较多的内存。如果 LIB 报告一个 no more Virtual memory 信息时，可以使用/NOE 或使用较少的模块去建立程序库。

（4）/NOI［GNORECASE］

告诉 LIB，当比较符号时，要保留大小写。如果你在结合许多程序库时，其中有一些是分大小写的程序库，LIB 会将输出的程序库也分大小写。你可以使用/IGN 去强定此选项使之不分大小写。

（5）/NOL［OGO］

不列出 LIB 版权信息。

（6）/P［AGESIZE］：number

指定新程序库或已存在的程序库 page 大小。number 是指定 page 大小的 byte 数，它必定是 2 的次方（16～32768），缺省值是 16 bytes。程序库的 page 大小设置各模块存储在程序库的边界。各模块在文件的位置（起始位置；即 LIB 所产生的列表文件中的偏移），是由文件开头算起的 page 大小的倍数。当建立一个程序库时，LIB 会建立一个存储每个名称位置的表。page 的大小也决定了 .LIB 文件的最大值。此限制是 number * 64KB。例如，/PAGE：32 限制了 .LIB 文件的大小至 2MB（32 * 65535 bytes）。然而对程序库中每个模块，平均有（num-

ber/2) bytes 的存储空间会被浪费。大部分的情况下，较小的 page 是较有利的，你应该使用较小的 page 大小，除非你需要在程序库中放置一个非常多的模块。

(7) /?

显示 LIB 命令行语法的简略接要。

3. 命令 (Commands)

此命令区段指定使用 LIB 执行程序库管理的五种运算：add、delete、replace、copy 和 move。你可以在命令行或 Operations：中直接使用。为了使用这个区段，你必须在模块或目的文件名称之后立即键入命令运算符。你可以在此区段以任何顺序指定超过一种运算。如果你将此区段留下空白（即不使用），LIB 不会对 oldlibrary 做任何改变。

如果你必须执行许多运算，你可以使用“&”去延伸运算列。你必须在模块名称或文件名称之后使用它，不可在运算符和名称之间使用。而在“&”键入之后，立即按 Enter 键，可再执行其余命令。

使用这些命令的路径和文件名不能包含“-”字符。

(1) Add Command (+)

使用“+”去建立一个程序文件、增加一个模块或结合程序库。格式：

+name

name 是目的文件或程序库文件的名称。如果没有指定子程序文件名，LIB 会假设 .OBJ。你可以指定完整路径和文件名。使用实例如下：

①建立一个新的程序库

可以使用“+”将一个或多个目的文件结合起来去建立一个新的程序库。将此新的程序库名称（文件名）指定在 oldlibrary 区段，然后再将要被加入的目的文件名称立即写在其后，并且每个名称之前都要有个“+”符号。下面的实例，LIB 是被指示去建立一个包含名称为 MORE 目的模块的 FIRST. LIB 程序库（假设 FIRST. LIB 不存在）：

LIB FIRST +MORE;

由于在 oldlibrary 区段之后使用了 commands 区段（此例为+MORE），所以 LIB 不会询问你是否要建立此新文件。加“;”也有同样的效果，更何况使用了两种，那更不会询问你确定是否要建立此新程序库。

②增加程序库模块

同样使用“+”去增加一个目的模块到程序库中。目的文件名要立即加在“+”之后，LIB 会增加此目的模块到程序库结尾。

LIB 会去除目的模块的驱动器名、路径和子程序文件名，只留下主文件名。这将变成在程序库中目的模块的名称。例如，如果一个目的文件 B:\CURSOR. OBJ 被增加到程序库中，此目的模块的名称就是 CURSOR。

下面的实例，LIB 被指示去增加 MORE 模块到一个已存在的程序库文件 FIRST. LIB 中：

LIB FIRST +MORE;

③结合程序库

使用情况和规则与前面所述的相同。只是当要加入的子程序文件名为 .LIB 时，你必须写上 .LIB 的子程序文件名，否则 LIB 会假设文件是一个目的模块，而去寻找子程序文件名为 .OBJ 的文件。

LIB 会增加原来的程序库到新的程序库尾端，注意用来被增加的程序库还是完整存在，LIB 只是执行拷贝这些模块而不会删除它们。

一旦你已将程序库相加在一起，你可以在 newlibrary 区段指定一个新的程序库名称，将前面已结合好的程序库存储在你指定的新程序库中。如果你省略了此区段（newlibrary），LIB 会将结合好的程序库存储在 oldlibrary 区段的程序库中。原来的程序库将以原来的主文件名再加上 .BAK 子程序文件名存储。.BAK 是备份文件（BACKUP）的意思。下面的实例将 DRAW. LIB 和 CHART. LIB 结合到一个文件名为 GRAPHICS. LIB 的程序库中。

LIB DRAW + CHARET. LIB, , GRAPHICS

DRAW 在 oldlibrary 区段所以不需加子程序文件名。CHART 为 .LIB 子程序文件名，所以要加上子程序文件名。GRAPHICS 在 newlibrary 栏所以也不用加子程序文件名 .LIB。

（2）Delete Command（—）

使用“—”从程序文件中删除一个目的模块。格式：

—name

name 是将被删除的模块名称（OBJ）。要注意不可以指定路径或子程序文件名，只能有一个名称，如 CURSOR。

下面的实例告诉 LIB 从 MATH. LIB 程序库中删除 FLOAT 模块：

LIB MATH —FLOAT;

（3）Replace Command（—+）

使用“—+”去取代程序库中一个模块。格式：

—+name

name 是将被取代的模块名称，同样不可有路径和子程序文件名。LIB 会先将指定的模块删除，然后再将此相同的模块附加在其后。此目的文件假设有 .OBJ 子程序文件名且是在目前的目录中。

下列三个实例是相同的，都是指示 LIB 取代 LANG. LIB 程序库中的 HEAP 模块。LIB 会从程序库中删除 HEAP. OBJ，再将 HEAP. OBJ 附加在程序库之后，就像一个新的模块在程序库一样。删除的运算总是会在加的运算之前被执行，并不依照出现的顺序决定运算的先后。

LIB LANG —+HEAP ;

LIB LANG —HEAP +HEAP ;

LIB LANG +HEAP —HEAP ;

（4）Copy Command（*）

使用“*”从程序库中拷贝一个模块到一个有相同名称的新建立的目的文件中。格式：

*name

name 是将被拷贝的模块名称。此模块还是会留在原来的程序库中。LIB 会使用此模块的主文件名再加上一个 .OBJ 子程序文件名来命名此目的文件，并放在目前的目录中。你不可以改变此文件名或位置，然而你可以稍后再将它改名（REN）或搬移至其它位置。

（5）Move Command（—*）

使用“—*”从程序库中搬移一个目的模块到一个目的文件。格式：

—*name

name 是将被搬移的模块名称。这个运算和使用“*”拷贝模块到一个目的文件中，然后

再使用“-”从程序库中删除此模块相同。

4. 交叉参考文件（Cross－Reference Listing）

交叉参考文件包含两个列表：

①按照字母顺序列出所有在程序库中的公用符号，每个符号的名称是遵循在原来模块中所定义的名称命名。

②列出在程序库中所有模块的位置（偏移）和大小。在每个模块名称之后，一样会按照字母顺序列出所有定义在此模块中的公用符号。可以在命令行的 listfile 区段或提示字符串 List file：之后指定列表文件名称。你也可以指定完整的路径和列表文件名称。如果你没有指定子程序文件名的话，缺省值 LIB 不会替你加上子程序文件名，若没有指定将不建立。

下面的实例建立了一个 LCROSS. PUB 列表文件。此实例只会检查 LANG. LIB 程序库，而不执行其它的工作。建议使用 .LIS 子程序文件名以与 LST 文件有所区别。

LIB LANG，LCROSS. PUB；

5. 输出程序库文件（Output Library）

在 newlibrary 区段或在提示字符串 Library name：之后指定一个被改变的程序库文件名。也可以使用完整的路径和文件名。如果你没有提供子程序文件名，缺省值并不会将其自动加上子程序。若此文件已存在则将被改变。所有在 commands 区段或在 Operations：之后所指定的运算结果都会在此文件中执行。LIB 会保留一个未改变和一个新改变的程序库，这是怕你执行的过程未完成时，终止了 LIB 的执行，而破坏原来的程序库文件。

①如果你在 newlibrary 区段指定一个新的程序库名称，所有修改的结果将存储在新的程序库中，并不会更动原来的程序库。

②如果你没有指定此区段，LIB 会将改变结果取代原来的程序库，原来的程序库将以 .BAK 子程序文件名存储。不管是哪一种方式，结果都会有两个版本：新版本和原来的版本。

8.7 建立程序库

前面我们之所以花这么多的篇幅详细说明，无非是希望读者能学会程序库的用法。程序的技巧可以下功夫去练习，但学习的方向要正确。因为编写汇编的程序实在是工程浩大，若您不懂得利用一些公用程序，而停留在每次编写都要花掉宝贵的休息时间，那作者真是太对不起大家了，那也违背我们让程序多做点、而人少做点的原则。

我们仔细观察一下前面的子程序都使用了 .model、.data、.code 和 END。这和我们前面写程序的习惯完全相同。而要注意的是对子程序千万不要定义 .stack 堆栈段，因为只能有一个堆栈段。还有就是无论主程序是以何种方式来开始与结束程序，记住子程序的 END 之后千万不要加任何程序的起始点，因为程序的起始点只能有一个，一般是在主程序中。

接下来我们赶紧将前面所写的子程序，结合起来建立一个程序库。首先要将所有的子程序都编译成 .OBJ 文件，接下来再利用 LIB 建立一个程序库。.ASM 文件是不能直接拿来建立 .LIB 文件的，相信大家都有这个共识。

利用 ML 来编译每个子程序，那也是相当麻烦的。我们共有 17 个子程序，那共要编译 17 次。我们可以利用一种叫做回应（response）文件方法。下面是 conio. prj 文件的内容，这是为使用 ML 编译所建的：

/c setvmode.asm cursrlen.asm gotoxy.asm getxy.asm setvpage.asm scrolwin.asm clrscr.asm getvmode.asm getche.asm getch.asm putchar.asm readkey.asm getstr.asm putstr.asm keystat.asm clearkey.asm writline.asm

各位一定要在 PWB 中建立此文件。执行 PWB conio.prj 和 PE3 的使用相类似。若此文件不存在会问你要建立否，答是“Yes”，按 Enter 键即可。将 conio.prj 的内容键入，再按 Shift+F_2 或在 File 的菜单中选 Save，再按 Alt+F_4 键或 File 菜单下的 Exit 退出。

下面是 conio.res 的内容，这是为使用 LIB 建立程序库所建的。同样要在 PWB 中建立：

conio－＋setvmode－＋cursrlen－＋gotoxy－＋getxy－＋setvpage－＋scrolwin－＋clrscr &－＋getvmode－＋getche－＋getch－＋putchar－＋readkey－＋getstr－＋putstr－＋keystat &－＋clearkey－－writline，conio.lis；

原本要在 DOS 下编译和使用 LIB 的执行情况如下：

①ML/C setvmode.asm cursrlen.asm gotoxy.asm....

②LIB conio－＋setvmode－＋cursrlen－＋gotoxy...

现在只要如下（记住键入"@"）：

①ML @conio.prj

②LIB @conio.res

相当的简短。执行①之后会产生 conio.prj 文件中所有 .ASM 文件的 .OBJ 文件。执行②之后会产生 conio.res 文件中所列的 conio.lib 与 conio.lis 两个文件。它的好处在于日后不管修改了几个子程序，只要再执行以上的步骤即可。若有新加入的子程序可要再修改 conio.prj 与 conio.res 两文件的内容。conio.res 中使用“－＋”的运算是避免日后再执行一次时，原有的子程序要先使用“－”删除才可再重新加入。

假设我们有一个主程序叫 test1.asm，执行时要引用 conio.lib，只要执行如下程序即可：

ML test1.asm conio.lib

记住结尾不要加“;”。

也可以利用 INCLUDELIB 伪指令。把 INCLUDELIB CONIO 写在程序中任何地方皆可，就可顺利引用此程序库。当然不可写在 END 之后，您应该知道它的原因。建议写在程序开头较好看。你故意写在程序后头，到时连自己都找不到那可就不好办了。

下面我们举个应用的例子（test.asm）。由于各子程序说明了几个变量，所以在主程序引用时要将他们说明为外部变量，当然子程序也要说明为外部才可。为了方便使用，我们建立一个包含文件（conio.h），以后要使用此子程序时只要记得把它 include 进来即可。哈！哈！，怎么那么像 C 语言。这是为了方便 C 语言用户的习惯，因为我也是 C 语言的拥护者。下面是 conio.h 的内容（不一定要在 PWB 中建立）：

```
EXTERN Setvmode: Proc, Cursrlen: Proc,
       Gotoxy: Proc, Getxy: Proc,
       Setvpage: Proc, Scrolwin: Proc,
       Clrscr: Proc, Getvmode: Proc,
       Getche: Proc, Getch: Proc,
       Putchar: Proc, Readkey: Proc,
       Getstr: Proc, Putstr: Proc,
```

```
        Keystat: Proc, Clearkey: Proc,
        Writline: Proc,
EXTERN  Row: BYTE, Column: BYTE, Char: BYTE, Show_Page: BYTE,
        Cursor_Top: BYTE, Cursor_Bottom: BYTE,
        Video_Attribute: BYTE, Video_Mode: BYTE,
        Up_Left_Corner: BYTE, Down_Right_Corner: BYTE,
        Scroll_Function_Number: BYTE, Scroll_Line_Number: BYTE
```

我们在 test.asm 中示范了许多子程序的应用。程序执行的过程首先将原屏幕的数据存储在 video_buffer 中，再清屏幕，这也是我们最常做的。接下来开 17 个窗口，并等待按键后再退出，在 DOS 下执行 ML test.asm 即可。下面是 test.asm 文件的内容。

Test.asm

```
.dosseg
.286
.model small
.stack 1024
        include conio.h
        includelib conio.lib
        VIDEO_SEGMENT   EQU 0B800h      ; 单色屏幕请改为 0B000h
        VIDEO_SIZE      EQU 80*25       ; 多使用 EQU 是个好习惯
        DELAY_SPEED     EQU 1000
.data
        video_buffer        WORD VIDEO_SIZE DUP (?)      ; 共 4000 bytes
        message             BYTE "Press any key to Quit ! $"
        cursor_position     WORD?

    .code
        main proc
            .STARTUP
                call  save_screen
                call  save_cursor
                call  clrscr
                call  scroll_windows
                mov row, 19                  ; 打出信息的位置 (x=36, y=19)
                mov column, 36
                call  gotoxy
                mov dx, offset message
                call  putstr
                call  clearkey
                call  getch
                call  restore_screen
                call  restore_cursor
            .EXIT
main endp
```

```
save _screen proc
        push ax
        push cx
        push si
        push di
        push es
          mov ax, VIDEO _ SEGMENT          ;要通过一个通用寄存器,不可
          mov es, ax                       ;直接 mov 到 ES 段寄存器
          mov di, offset video _ buffer
          mov cx, VIDEO _ SIZE             执行循环的次数指定在 cx
          mov si, 0                        ;sub si, si 或 and si, 0 也可
          save _ screen _label:            ;标记命名多利用子程序名称
                    mov ax, ES:[si]        ;段强定为能指到屏幕地址
                    mov [di], ax
                    add si, 2              ;是加 2,因为包含 ASCII 字符
                    add di, 2              ;与属性
                  loop save _ screen _ label
        pop es                             ;循环的使用请参阅第 9 章
        pop di
        pop si
        pop cx
        pop ax
    ret
save _ screen endp

save _ cursor proc
      mov show _ page, 0
      call getxy
      mov cursor _ top, ch
      mov cursor _ bottom, cl
      mov byte ptr cursor _ position, dl
      mov byte ptr cursor _ position + 1, dh
    ret
save _ cursor endp

restore _ screen proc
        push ax
        push cx
        push si
        push di
        push es
          mov ax, VIDEO _ SEGMENT
```

```
    mov es, ax
    mov si, offset video _ buffer
    mov cx, VIDEO _ SIZE
    mov di, 0
    restore _ screen _ label:
                    mov ax, [si]
                    mov ES: [di], ax
                    add si, 2
                    add di, 2
                loop restore _ screen _ label
pop ax
pop cx
pop si
pop di
pop es
ret
restore _ screen endp
restore _ cursor proc
push dx
mov dl, byte ptr cursor _ position
mov dh, byte ptr cursor _ position+1
mov row, dh
mov column, dl
call gotoxy
call cursrlen
pop dx
ret
restore _ cursor endp
scroll _ windows proc
push cx
mov scroll _ function _ number, 07h
mov scroll _ line _ number, 7
mov video _ attribute, 1Eh
mov up _ left _ corner, 0
mov down _ right _ corner, 0620h
mov cx, 17
scroll _label:
call scrolwin
call delay
add up _ left _ corner, 0102h
add down _ right _ corner, 102h
add video _ attribute, 10h
```

```
            loop scroll _ label
          pop cx
        ret
scroll _ windows endp

delay proc
        push ax
        push cx                                    ;请参阅第 9 章
          mov ax, 0
          .repeat                                  ;.repeat 与 .while 伪指令
            mov cx, 0                              ;可供互相网状使用
            .while (cx< =DELAY _ SPEED)
                inc cx
            .endw
            inc ax
          .until (ax> =DELAY _ SPEED)
            pop cx
            pop ax
        ret
    delay endp
end
```

第9章　程序流程

一开始我们介绍在控制程序流程指令的讨论中所需要的工具（标记与标志）。循环指令一直是语言所提供最强大的功能之一。对于一些重复性的工作如何利用循环以减少程序设计的时间及工作份量，也是本章的重点之一。

一些简单的程序都是按照程序中指令出现的顺序执行，但在许多较复杂的大型程序中，有时可能基于某些原因，而改变程序的流程，去执行某些不同片断的叙述。汇编语言提供了许多无条件及有条件的转移指令，以使我们能有较高的权力去控制程序的执行流程。

第一个章节介绍在程序中由一个点跳至另一个点，解释 MASM 6.1 如何在某些情况下简化无条件和条件转移。这样你将不必去指明每一个属性，也会描述你可以使用去测试条件转移的指令。

下一个章节描述循环结构的重复行为或计算条件，也会讨论一些 MASM 的伪指令，例如：.WHILE 和 .REPEAT，它可为你产生适合的比较、循环和转移指令，也可以产生转移指令的 .IF、.ELSE 和 .ELSEIF 伪指令。

9.1　转移

转移是改变程序执行从一个位置到另一个位置最直接的方式。站在处理器的角度，转移的执行只是借助改变 IP（Instruction Pointer；指令指针）寄存器的值到目的地的偏移。对于远程转移，则还要改变 CS 寄存器到一个新的段地址。转移指令又可分为条件和无条件两种。

9.2　在代码段中的标记

可以在代码段中使用标记去做为一个记号，也就是说可以在程序中借助使它跳至某一个标记去控制程序的流程。

使用 LABEL 伪指令去定义一个标记，格式如下：

```
name LABEL type
```

在代码段中使用标记有两种型式：NEAR 和 FAR。如下：

```
L1      LABEL NEAR
ENTRY LABEL FAR
```

第一个例子定义一个名为 L1，型式为 NEAR 的标记。第二个例子定义一个名为 ENTRY，型式为 FAR 的标记。

代码段的标记可做为一个地址的记号。一个近程标记可代表只可以从相同段中参考的地址记号。远程标记则是可以从另一个段参考的地址记号。

大部分的标记都是近程的。换言之，总是在相同的段内转移。尽量保持在一个子程序中执行转移的操作是一个良好的程序设计习惯。远程标记则是较少用的，除非是在大而复杂的

程序中有许多段时，为了要转移至另一个段中的子程序时才需要。

要了解近程与远程标记的区别的最好方法，就是去了解编译程序如何去处理它们。编译程序会为所有出现的标记建立一个表格，其中包含了名称、地址及类型。

对于一个近程标记，编译程序使用一个包含偏移的地址（这需要 1 word）。对于一个远程标记，编译程序使用一组完整地址：一个段地址和一个偏移地址（这需要 2 words），这是因为要跳至另一个段处理器需要一个段地址和一个偏移。

因为大部分的标记都属于近程，编译程序允许你用下列缩写的格式定义一个近程标记：

name：

例如：L1：

READ _ CHAR：

换言之，编译程序会将一个以“：”做结尾的名称视为一个近程标记。因此 L1：是和 L1 LABEL NEAR 一样的。很明显的，这种缩写格式是简单和受欢迎的。

当你使用缩写型式去定义一个标记时，你可以在同一列放置一个指令。如下：

TESTBX：CMP BX，0

然而让标记独立在一列应该是一个比较好的习惯，因为较易读。如下：

TESTBX：

CMP BX，0

标记的名称最长可以到 31 个字符。这样你可以较明确地描述此标记的意义，例如：

PRINT _ STRING：

PROCESS _ NEXT _ NUMBER：

READ _ NEXT _ FILE _ RECORD：

至于到底要使用较长或较短的标记名称，下面有两个指导建议：

对一群有相关性的标记且是在相同的区段，短标记名称较适合。

如需有明确的标记意义指明时，使用较长、有描述意义的标记名称较适合。

例如每个子程序可能都需要设计一个退出子程序的标记，你可以子程序名称再加上 FINISHED，来代表退出子程序的标记，这样就不会为了设计一个代表子程序结尾而伤脑筋，而且有可读性、一致性。实例如下：

FINISHED _ DISPLAY：

FINISHED _ TEST _ INPUT：

FINISHED _ READ _ FILE _ RECORD：

9.3 标志寄存器

标志寄存器是和其它寄存器不同的。我们可以针对其中的某位处理。一个标志值（0 或 1），可被使用去代表一个特别的情况。如状态标志描述由某些指令所产生的结果，控制标志控制处理器的运算。

每个状态标志和控制标志都有一个缩写名称（表 9-1）。因为每个标志是一个 bit，它的值不是 0 就是1。当我们给定标志一个值为 1 时，称为设置标志；当给定值为 0 时，称为清除标志。

标志在标志寄存器中的位置列于图 9-1，其中 15、5、3、和 1 是未使用的位，14、13、12 只使用在保护模式。我们很少需要去知道标志真正的位置。

表 9-1　状态和控制标志的名称及缩写

状　态　标　志	
缩　写	名　称
AF	Auxiliary carry flag
CF	Carry flag
OF	Overflow flag
PF	Parity flag
SF	Sign flag
ZF	Zero flag
控　制　标　志	
缩　写	名　称
DF	Direction flag
IF	Interrupt flag
TF	Trap flag

标志在标志寄存器的位置

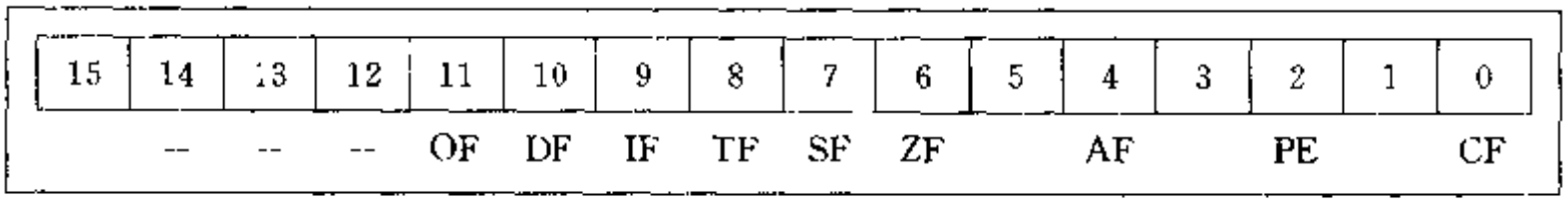


图 9-1　标志在标志寄存器的位置

9.3.1　状态标志

状态标志可由许多指令修改，且影响可能不止一个标志。有许多状态标志是很少使用的，重要的使用介绍如下：

- Zero flag：ZF 标志被设置表示运算的结果为 0。
- Sign flag：SF 标志被设置表示运算的结果是负数。
- Overflow flag：OF 使用在有符号数。OF 被设置表示运算的结果太大无法存储在目的操作数。

- Carry flag：CF 使用在无符号数。CF 被设置表示运算的结果太大无法存储在目的操作数。（例如相加的结果太大产生进位）

Auxiliary carry flag：AF 是使用在十进制算术。如十进制加减后需要调整时 AF 被设为 1。事实上，当你使用十进制算术时，AF 会自动处理设置，所以你不必真正自己去处理。

- Parity flag：当指令运算的结果，位值包含偶数个 1 时，PF 会被设置为 1。如果是包含奇数个 1，则 PF 被清为 0。此标志最典型的用法是当有数据传输时检查之用。（如果你不做这类的工作，到是可以忘记此标志。）

进位标志有个很重要的使用技巧，就是可以作为设计子程序是否成功的终止。如果子程序结束没有错误，可以在 return 之前清除 CF。如果有错误被检测到，可以在子程序 return 之

前，设置 CF 为 1。这样调用的程序可以通过检查 CF 去知道是否有错误发生。而子程序也可通过 AX 或 AL 传回错误码。

9.3.2 修改状态标志指令：STC、CLC 和 CMC

大致来说没有修改状态标志的需要，只要去检查即可。但是有时你还是需要去修改进位标志（CF）。由于这个原因，处理器提供了三个指令可修改状态标志值。如表 9-2：

表 9-2

STC	(set carry flag)	设置 CF 为 1
CLC	(clear carry flag)	清除 CF 为 0
CMC	(complement carry flag)	反转 CF 的值

这些指令有两种使用情况：

①你可以使用 CF 去作子程序的检测之用，如前面所述。

②某些循环移位指令（如 RCL、RCR）也使用 CF 当作部分运算。你可以在使用 RCL 和 RCR 指令之前改变 CF 为一个特殊值。对于其它的状态标志是没有简单的方法去改变其值的。

9.3.3 控制标志

控制标志可以控制处理器的运算。状态标志可以表示有关指令执行之后的信息，而控制标志可以修改指令还未执行之前的行为。控制标志的使用说明如下：

· 方向标志（DF）：DF 一般使用在字符串指令中。（注意字符串就是一连串的 bytes 或 words。）有些字符串指令你必须指明字符串将被处理的开头地址和长度。如果 DF=0，字符串从左往右处理；DF=1，表从右往左处理。

· 中断标志（IF）：内部、外部装置和某些程序要得到处理器注意的方式是重要的。可以使用装置或程序内的系统暂时终止处理器。（当这个情况发生时，我们称此装置或程序送出中断。）完整的中断请参阅附录。

· 单步标志（IF）：TF 允许程序去在每个指令执行之后放置一个子程序准备被调用。这是为了让调试（Debugger）使用。可以让执行的程序一次执行一个指令，而让两个指令之间暂停一下。

如果 TF=1，处理器会在每个指令执行之后调用一个特别的子程序（通过产生一个中断）；如果 TF=0，处理器运算正常。除非你自己编写调试（Debugger），否则你不需要去设置这个标志。

9.3.4 修改控制标志指令：STD、CLD、STI 和 CLI

有两个指令可以修改 DF，两个指令可以修改 IF（表 9-3）：

表 9-3

STD	set direction flag	设置 DF 为 1
CLD	clear direction flag	清除 DF 为 0
STI	set interrupt flag	设置 IF 为 1
CLI	clear interrupt flag	清除 IF 为 0

在你每次使用字符串指令之前都必须设置或清除 DF。你不需要去设置 IF，除非是要编写处理外部装置的程序。对于 TF 你没有必要去修改，因此，没有直接修改 TF 的指令。

9.4 条件转移指令

有许多指令可以让你使用去影响程序的流程，而这些指令大部分都是条件转移指令。有两个处理的步骤：

（1）首先测试条件。

（2）如果条件成立（是真）则执行转移；如果条件不成立（是假）则继续执行下面的语句。因此，任何可以设置或清除标志的语句都可作为条件转移的测试基准。而一个条件转移指令之后可以接一个包含目的地址的操作数。不可以使用指针值当成无条件转移的目的地。正常处理器执行指令是一个接着一个执行，而一个转移指令可以告诉处理器去继续执行在一个通过一个特别标记指定的地址。事实上，控制权会转移至标记下的下一个指令。例如说程序中包含这个语句：

```
L1:
    mov ax, 0
```

如果处理器执行一个转移至标记 L1，则 L1 下一个语句将被执行：

'mov ax, 0'。

如果一个指定的条件是真，则条件转移将被执行转移。条件转移指令的格式如下：

```
Jxxx labell
```

xxx 是一个特别条件的缩写。如果条件成立（是真），处理器会继续执行标记后的第一个语句。如果条件不成立（是假），则继续执行下一个语句。

这里有一个实例（包含许多语句）：

```
; if CX is not equal to 0, set it to 10
    JCXZ L1
    MOV DX, 10
L1:
```

如果 CX 寄存器的值是 0，JCXZ（稍后会讨论）指令会跳至标记 L1。（在这个情况，转移的条件是 CX 是否等于 0）。

如果转移发生，处理器将跳过 MOV 指令。如果 CX 不等于 0，处理器将不会转移，而会执行 MOV 指令。

有 10 种条件转移是依照标志值是否成立。有两种条件转移是依照 CX 寄存器的值。除了奇偶标志（parity flag），所有的状态标志都有两种条件转移指令，一个是如果标志被设置（1）则转移，另一个是如果标志值被清除（0）则转移。这些指令（如表 9-4 所示）都有一个缩写“J”开头。

两个 PF 转移有相同的 opcode。JPE（E 是代表“EVEN”，偶数的意思）和 JP 相同；JPO（O 是代表“ODD”，奇数的意思）和 JNO 相同。你可以依照你的需要使用，不过很少使用 PF 标志。

注意，并没有条件转移会测试 AF；使用 AF 标志的指令（十进制算术指令）会自动处理，并不需要你去设置、清除或测试 AF 标志。

只有两个条件转移会测试寄存器。由名称就可猜到，这个寄存器是CX或ECX，只有当CX或ECX等于0，此指令（JCXZ、JECXZ）才会转移。

之所以会测试CX值，是因为有许多重复运算的指令，重复的次数被存储在CX，就如同一个计数器（counter）。提供此种测试CX的指令是相当有用的。

表9-4 条件转移指令

指令（Opcode）	条件
JC/JB. JNAE	CF=1
JNC/JNB/JAE	CF=0
JBE/JNA	CF=1或ZF=1
JA/JNBE	CF=0和ZF=0
JE/JZ	ZF=1
JNE/JNZ	ZF=0
JL/JNGE	SF≠OF
JGE/JNL	SF=OF
JLE/JNG	ZF=1或SF≠OF
JG/JNLE	ZF=0和SF=OF
JS	SF=1
JNS	SF=0
JO	OF=1
JNO	OF=0
JP/JPE	PF=1（even parity）
JNP/JPO	PF=0（odd parity）

9.4.1 CMP指令

前面章节所学到的条件转移指令是直接检查状态标志（或CX）。然而大部分的时间，你可能需要比较两个数来决定条件转移是否成立。

CMP指令是测试条件转移最常见的方式。它可以比较两个值，但不改变他们，只是按照比较的结果去设置或清除处理器的标志。

本质上CMP指令和SUB指令相同（除了CMP指令不会改变目的操作数），但两个指令都会按照减法的结果去设置标志。

CMP op1，op2

CMP（compare）指令总是先比较两操作数的大小然后再执行之后的条件转移指令。下面有一些合法的实例使用如下：

- 两个寄存器比较：CMP AX，BX
- 一个寄存器与一个变量的比较：CMP AX，TOTAL

- 一个寄存器与一个立即数的比较：CMP AX，0
- 一个变量与一个寄存器的比较：CMP TOTAL，AX
- 一个变量与一个立即数的比较：CMP TOTAL，0

你不可以直接比较两个内存变量。如果需要的话，你可以先 copy 一个至寄存器再比较。有两个较重要的限制：

（1）你应该总是比较两个有符号数或无符号数，但不可混用。

（2）两个操作数必须是相同的大小，不是 bytes 就是 words。

CMP 指令事实上是将两个操作数都当做数值对待。工作的原理是将第一个操作数（op1）减去第二个操作数（op2），然后再设置相关的标志值，供接下来的指令判断之用。执行完 CMP 指令之后，两操作数本身并不改变。

如果你要比较两个字符请使用 CMPSB 与 CMPSW 指令，如果是要比较单独的位，可使用 TEST 指令。

在 CMP 指令之后可以使用的条件转移分为两组，一组适用于有符号数（表 9-5），另一组适用于无符号数（表 9-6）。

前面曾经讨论过，bytes 或 words 值，可以存储为有符号数或为无符号数，它是根据要使用所需要的类型而定。你可以比较有符号或无符号的值，但必须选择相对的条件转移去反应正确的值。

表 9-5　使用在有符号数的条件转移指令

指令（opcode）	意　　义
JL（JNGE）	如果小于则转移（不大于或等于）
JG（JNLE）	如果大于则转移（不小于或等于）
JLE（JNG）	如果小于或等于则转移（不大于）
JGE（JNL）	如果大于或等于则转移（不小于）
JE	如果等于则转移
JNE	如果不等于则转移

表 9-6　使用在无符号数的条件转移指令

指令（opcode）	意　　义
JB（JNAE）	如果小于则转移（不大于或等于）
JA（JNBE）	如果大于则转移（不小于或等于）
JBE（JNA）	如果小于或等于则转移（不大于）
JAE（JNB）	如果大于或等于则转移（不小于）
JE	如果等于则转移
JNE	如果不等于则转移

各缩写符号所代表的意义如下：

缩写符号	意　　义
N	Not
E	Equal
G	Greater than
L	Less than
B	Below
A	Above

在CMP指令之后，正常都会使用一个条件转移指令去测试比较的结果。注意，有许多指令都有相同的opcode。在这个情况下，你可以使用任一个指令而使得你的程序较有意义。例如，JL（Jump if less than）是相等于JNGE（Jump if not greater than or equal）。编译程序提供你一个方便的选择。

有两个重点是需要了解的：

（1）处理器会针对有符号数或无符号数检查标志值，因此你所使用的指令必须告诉处理器要检查哪一个类型（有符号或无符号数）。两种指令逻辑上是相同的。记住要用“less”或“greater”表存取有符号数；用“below”或“above”存取无符号数。

这些指令检测“equal”和“not equal”对两种数字类型是相同的。

（2）由条件转移指令可以知道前面CMP指令所使用的两个操作数为有符号数或无符号数。因此JGE指令假设先前是两个有符号数的比较。如果发现先前第一个有符号数大于或等于第二个有符号数则转移。而JAE指令也有相同的影响，唯一的不同是假设前面CMP指令的操作数是无符号数。这里有许多使用CMP条件转移的实例：

（1）如果一个有符号数变量TOTAL小于0，则会跳过许多指令：

```
        CMP TOTAL，0
        JL    L1
；——将被跳过的指令——
    L1：
```

如果你需要也可以使用JNGE代替JL：

```
        CMP  TOTAL，0
        JNGE L1
；——将被跳过的指令——
    L1：
```

（2）第二个实例，如果一个无符号数变量SUM小于在AX中的无符号数，则会跳过许多指令。因为两个数都是无符号数，所以使用JB（below）代替JL（less than）：

```
        CMP SUM，AX
        JB    L2
——将被跳过的指令——
    L2：
```

(3) 第三个实例，如果 DI 大于或等于变量 VALUE，会跳过许多指令。假设 DI 与 VALUE 变量存储有符号数：

```
        CMP DI，VALUE
        JGE  L3
；——将被跳过的指令——
    L3：
```

(4) 最后一个实例，如果 AH 是大于或等于 DH 则会转移。假设 AH 和 DH 存储无符号数。(所以使用 JAE 代替 JGE)

```
        CMP AH，DH
        JAE  L4
；——将被跳过的指令
    L4：
```

9.4.2 以位指定为根据的转移

在单一值中的独立位指定也可提供作为条件转移的根据。TEST 指令可以测试在一个操作数中指定位是 ON 或 OFF（设置或清除）和 ZF。TEST 指令和 AND 指令是相同的（除了 TEST 指令不会改变任何操作数）。下列的实例说明 TEST 的应用：

```
    .DATA
          bits BYTE ?
    .CODE
    .
    .
; if bit 2 or bit 4 is set，then call task _a
    test bits，10100b
    jz    skip1              ; If 2 or 4 is set
    call  task _a            ; Then call task _a
skip1：                      ; Jump taken
    .
    .
    .
; if bits 2 and 4 are clear，then call task _b
    test bits，10100b        ; If 2 and 4 are clear
    jnz  skip2
    call task _b             ; Then call task _b
skip2：                      ; Jump taken
```

9.4.3 以零为根据的转移

程序通常需要以某个寄存器内是否包含 0 为根据去执行转移。我们知道 JCXZ 指令如何依照在 CX 寄存器的值去转移。你也可以使用 OR 指令同样有效地去测试在其它数据寄存器值是否为 0。例如下列的实例测试 BX 是否为 0：

```
    or    bx，bx     ；Is BX=0？(较快)
```

```
jz    is _ zero    ; Jump if so
```

功能基本上是相等的：

```
cmp  bx，0       ; Is BX=0？（较慢）
je    is _ zero    ; Jump if so
```

但是产生较小和较快的程序码，因为它不必使用一个立即数当做操作数。相同的技巧也可让你测试寄存器的符号位（sign bit）：

```
or    dx，dx      ; Is DX sign bit set ?
js    sign _ set   ; Jump if so
```

9.4.4 转移扩展（jump-extending）

不像无条件转移，条件转移不能存取一个超过 128 bytes 远的标记。例如，下列的语句只要目标（target）是在 128 bytes 的范围内就是合法：

```
; Jump to target less than 128 bytes away
    jz target ; If previous operation resulted
              ; in zewro，jump to target
```

然而如果 target 太远，下列的结果将可以形成一个较远的转移。注意此结果逻辑上和先前的实例是相等的：

```
; Jump to distant target previously required two steps
jnz  skip        ; If previous operation result is
                 ; NOT zero，jump to "skip"
jmp  targe       ; Otherwise，jump to target
```

MASM 会自动为你扩展转移（jump-extending）。如果你的条件转移目标是一个超过 128 bytes 远的标记，MASM 会使用一个无条件转移重写此指令，以确使转移可以到达它的目的地。如果 target 是在 128 bytes 的范围内，编译程序还是会将指令编码成 jz target，否则 MASM 会产生两个替代的指令：

```
jne   $ + 2 + (length in bytes of the next instruction)
jmp  NEAR   PTR   target
```

如果你使用 NEAR PTR，FAR PTR，或 SHORT 指定距离，编译程序产生相同的结果：

```
jz    NEAR   PTR   target
```

变成

```
jne   $ + 5
jmp  NEAR   PTR   target
```

即使 target 小于 128 bytes。

MASM 缺省值会自动转移扩展，但你也可以使用 OPTION 伪指令的 NOLJMP 型式关掉此缺省值。

因为 JCXZ 和 JECXZ 指令没有逻辑的否定，所以转移扩展并不适用，因此距离总是 SHORT。

目的操作数的大小和距离决定对于条件或无条件转移至外部或不同段的目的编码。转移扩展和最佳化的特点不能应用在此种情况。

在 80386/486 条件转移可以到 32KB 远，所以转移扩展只发生在目的地比距离远时。

9.5 无条件转移指令：JMP

无条件转移指令不需要依照某一个条件是否成立，它总是会执行转移。无条件转移指令JMP的语法如下：

JMP label

下面有许多实例：

JMP L1

JMP FINISH_DISPLAY

JMP TABEL [SI]

使用JMP指令有两个目的：

(1) 当某一个条件成立时它可以跳退出一些不应该执行的指令。

(2) 可以往回跳以形成一个循环。

转移指令使用地址的规则

使用无条件转移指令（JMP），你可以在相同的代码段（near）或另一个代码段（far）指定任何地址。

条件转移指令较有限制。条件转移只可以跳至一个近程标记（near label），也就是此标记必须是在此指令的127 bytes的范围内。

为了提高效率，处理器在将条件转移机器语言翻译时，会将标记的地址编码在1 byte内。此byte存储一个有符号数，表示从下一个指令起与标记的距离（bytes）。（使用一个有符号数是因为转移可以往前或往后。）因为一个byte可以存储一个有符号数（-128～127），在条件转移指令内的标记必须在此范围内（bytes）。

一个好程序应该避免一个长程转移。然而你还是经常会遇到无法避免的长程转移。比如我们可能会利用进位标志去决定某些将要处理的数据是否正确。例如在子程序某一个特别点去检查数据是否有效；如果数据有效则CF清为0，若数据无效则CF设为1。如果数据无效，程序应设计为立即跳至子程序中的某一个特别区段（如标记INVALID_DATA）。

这里有一个实行比较和转移的指令：

```
; if data is invalid (CF=1) then execute error section
        JC invalid_data
```

除非这个标记invalid_data距离JC指令超过128 bytes，否则这是个很好的实例。如果超过128 bytes，有两个方案可解决：①重写这个子程序以使这个标记较接近此转移；②使用无条件转移指令。

如果发现不可能去重写此子程序的逻辑，你必须去使用无条件转移。你需要有一个条件转移测试一个相反的情况。此条件转移会跳过无条件转移指令，而跳到一个需要的标记。

这里有最好的例子说明。下列语句逻辑上和上一个例子相同：

```
; if data is invalid (CF=1) then execute error section
        JNC   L1
        JMP   INVALID_DATA
    L1:
```

然而，因为转移至INVLID _DATA 是由 JMP 指令所做，所以此标记可以超过 128 bytes 远。当然只有当绝对需要时才使用此方法。

转移指令是强而有力的，它可以在任何时间转移至任何地方（几乎）。但这也可能造成超过你所想像的麻烦。一般在两个重要的情况下使用转移：

①可以使用转移去实行一些相同或重复的结构，例如：

```
; if AX=0 then add 5 to BX, else add 10 to CX
        CMP AX, 0
        JE  L1
        JNE L2
    L1:
        ADD BX, 5
        JMP L3
    L2:
        ADD CX, 10
    L3:
```

②当一个程序检测到有例外情况时，可能必须立即跳至一个特别的子程序或到目前子程序的一个特别部分。例如，一个读取和处理数据的子程序应该要确定数据是否有效。如果无效，此子程序应该要跳过所有处理语句，并且将控制权转移至错误处理区段。

无论是条件或无条件，除非是为了形成一个循环，不要尝试往回跳。

9.6 循环（LOOP）

循环(LOOP)是一连串重复执行的指令。处理器提供一群指令使我们可以容易地实行循环。

第一个介绍的指令是 LOOP。LOOP 指令建立一个可以执行指定决定的循环。例如说，你有一个表格包含 100 个数字。如果你想要这些数字的总和，你应该需要一个可以执行 100 次的循环。此 LOOP 指令的格式如下：

```
LOOP label
```

和条件转移一样，标记必须在这个指令的 127 bytes 范围之内。通常这不是一个问题。

此 LOOP 指令使用 CX 寄存器当作计数器，CX 可作为追从循环已被执行过的次数。记住，CX 是“count”寄存器）开始前你要将循环执行的次数先载入 CX。实例如下：

```
    MOV CX, loop _count
lalbe:
---statements to be executed---
    LOOP labell
```

LOOP 指令执行时每次都会将 CX 减 1，如果 CX 不等于 0，LOOP 会跳至指定的标记。下面是一个将执行 100 次的循环：

```
    MOV CX, 100
L1:
---statements to be executed---
```

```
    LOOP L1
```

这里有个实例可以加 100 个数字。这些数字定义如下：

```
TABLE DW 100 DUP (?)
```

这些值由程序设置。这里有个循环可以加 100 个数字，总和在 SUM。(一般累加数字的结果都存储在 AX，因此名称为 “accumulator”。)

```
; set AX to the sum of 100 numbers, stroed as words in TABLE
        MOV AX, 0
        MOV SI, 0
        MOV CX, 100
L1:
        ADD AX, TABLE [SI]
        ADD SI, 2
        LOOP L1
```

这个实例使用三个寄存器：CX 被使用当作 LOOP 指令的计数器，SI 被使用当作存取在 TABLE 中的数字的索引，AX 被使用去累加总和。

计数器的工作如下：CX 的初值被置为 100，然后循环内的语句被执行（两个 ADD 指令)。LOOP 指令将 CX 减 1 并且检查此结果。因为 CX 是 99(不等于 0)，所以 LOOP 跳至 L1。因此这两个 ADD 指令被执行第二次。此循环会继续直到 CX 值为 0。ADD 指令之后将被执行 100 次。

索引工作如下；SI 初值被置为 0。每次经过循环，SI 会累加 2。因此 SI 值为 0，2，4 等等。第一个 ADD 指令使用 SI 做为变址寄存器，因此第二个操作数的值的 TABLE+0，TABLE+2，TABLE+4 等等。

累加器的工作如下：AX 初值为 0。每次经过循环，一个新的值将被加到 AX。因此当循环完成时，AX 将包含此总和。

下面有几个 LOOP 指令应用的实例，你将会发现循环的作用：

我们可以将要重复工作的次数指定在 CX 中，以节省设计的时间。下面，我们将字符 A′ 重复打印 25×80 次。注意由于 ‘mov ah，2’ 与 ‘mov d1,′A′’ 是相同的工作。所以可将他们搬至 next 标记之前只写一次即可。不必要浪费处理器每次都要执行搬移的时间：

```
    mov  cx, 25 * 80
next:
    mov  ah, 2
    mov  d1,'A'
    int  21h
    loop next
```

可改写如下：

```
    mov  cx, 25 * 80
    mov  ah, 2
    mov  d1,'A'
next:
```

```
        int     21h
        loop    next
```

※ 如果'A'改为空字符''，不也达到清屏幕的功能。不过这并没有比较好用或较快。

打印字符串也是程序设计中相当频繁的工作。例如要做好用户界面，打印提示字符串。前面介绍过两个打印字符串的方法（INT 21H 的第 09H 及第 40H 功能）。我们也可以利用循环来做打印字符串的工作：

```
.data
      str       byte 'This is a string example.'
      str_len   word $ - str
.code

putchar:
          mov  ah, 2
          mov  dl, [si]
          int  21h
          inc  si
          loop putchar
```

下面 CH9_2.ASM 示范打印 ASCII 字符 32～255。因为 32 以下的字符在 DOS 的解读下许多都是控制字符，为不使画面被破坏，不予打印。可利用 BIOS INT 10H 第 09H 及 OAH 功能。则不会有控制字符出现。

CH9_2.ASM

```
; 打印 ASCII 字符 32～255
.DOSSEG
.286
.MODEL SMALL
.STACK 1024
.DATA
.CODE
        main proc
            .STARTUP
              mov  cx, 255-31
              mov  dl, 32
              mov  ah, 2
        display:
               int  21h
               inc  dl
               loop display
            .EXIT
        main endp
END
```

我们可以利用 LOOP 与 CX 的微妙关系去求 1 加到 100 的总和。

```
        mov   ax, 0
        mov   cx, 100
again:
        add   ax, cx
        loop  again   ; 总和在 AX
```

在 C 语言中常使用 delay 去延迟程序的执行速度，在此也可利用 LOOP 指令去做。

```
        mov   cx; 32767   ; 最多只能到 32767
delay: loop delay
```

如果在 LOOP 指令执行之前(就是还未将 CX 减 1),CX 就等于 0,则第一次执行完 LOOP 之后，CX 等于－1 (OFFFFh)，也就是 LOOP 会执行 65535 次。在 LOOP 执行期间，尽量不要更改 CX 之值，除非你能控制它的执行情况。

网状循环也是在高级语言中经常见到的应用。我们可想到内循环可能会改变外循环的 CX 寄存器（计数器）的值。(如果你想不到，应该是你的功力不够，应再多练习写程序。你可将在高级语言学习时的实例拿来用汇编再写一次，必可增强你汇编的功力。这不是作者所能替你做的，望你多加油)。有一个技巧就是进入内循环之前先把 CX，push 入堆栈，退出内循环之后再 pop 出来。或内外循环各用一个变量记录。方法如下：

```
        mov   cx, out _counter
outer:
        .
        .
        mov   out _counter, cx   ; 可改为 push cx
        mov   cx, in _counter
        inner:
        .
        .
        loop  inner
        mov   cx, out _counter   ; 可改为 pop cx
        loop  outer
```

out _counter 变量表示外循环的执行次数，in _counter 变量表示内循环的执行次数。实例 CH9 _2. ASM、CH9 _3. ASM 如下：

CH9 _2. ASM

```
.DOSSEG
.286
.MODEL SMALL
.STACK 1024
PUTCHAR MACRO char
        mov ah, 2
        mov dl, char
```

```
            int   21h
 ENDM
.DATA
        in _ counter    WORD 78
        out _ counter   WORD 10
        number _ char BYTE 47
        Eng _ letter    BYTE 'A'
.CODE
         main proc
              . STARTUP
             ;
                mov cx, out _ counter
         outer:
                inc number _ char
                putchar number _ char
                putchar ')'
                mov out _ counter, cx   ; 可改为 PUSH CX
                mov cx, in _ counter
                inner:
                      putchar Eng _ letter
                      loop inner
                inc Eng _ letter
                mov cx, out _ counter     ; 可改为 POP CX
                loop outer
             ;
              .EXIT
        main endp
END
```

CH9 _ 3. ASM

```
  TITLE  将大写字符转换成小写字符
.DOSSEG
.286
.MODEL SMALL
.STACK 1024
.DATA
        letter _ str BYTE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 0
        len          WORD $-letter _ str
.CODE
        main proc
             . STARTUP
            ;
              mov ah, 2
```

```
            mov bx, 0
            mov cx, len
            M1:
                mov dl, letter_str [bx]
                or dl, 00100000B      ; 00100000B=20H
                int 21h
                inc bx
                loop M1
        ;
         .EXIT
    main endp
END
```

BX 索引值也可改为 SI 或 DI。又'a'－'A'＝32D＝20H，所以要将'A'转变为'a'需加 32。当然也可改为 'ADD dl，32'，不过用 'OR dl，32'，较快。

比较循环：LOOPE 和 LOOPNE

LOOP 指令是依照在 CX 的值建立循环。有时候也需要依照比较的结果去建立循环。LOOPE 和 LOOPNE 指令提供这个服务。

就像 LOOP 指令，LOOPE 和 LOOPNE 指令会将 CX 减 1，并且只有当 CX 不等于 0 才执行转移。然而 LOOPE 和 LOOPNE 指令还可利用另一个条件。LOOPE 还需要先前比较的两个操作数相等；LOOPNE 需要两操作数不相等。

例如你要在有 100 个数字的表格中搜寻第一个值为－1 的数。如果有一个比较发现有个值为－1，此循环就必须立即终止。这里有一个实例执行此循环。TABLE 是一个包含有 100 个（words）有符号数的表格，定义如下：

```
TABLE   DW   100   DUP   (?)
```

表格的数 字由程序设置。下列的循环会依照顺序去测试每一个数是否为－1。如果发现循环将终止；如果没有此循环将继续直到所有的数都被检查。

```
; search a table of 100 words for the value -1
        MOV     SI, -2
        MOV     CX, 100
    L1:
        ADD     SI, 2
        CMP     TABLE [SI], -1
        LOOPNE L1
```

如果你和上一个实例比较，你可以发现 SI 寄存器的初值不同。在这个实例 SI 初值为－2，原因是 ADD 指令必须要在比较之前，（如果 ADD 在 CMP 和 LOOPNE 之间，则可能会改变标志）因为你必须确定 CMP 第一次将被执行且 SI 值为 0，因此你必须确定使 SI 的初值为－2。

如果你是要搜寻第一个不为－1 的值，你必须使用 LOOPE 代替 LOOPNE。如下：

```
; search a table of 100 words for a value not equal to -1
```

```
        MOV     SI，－2
        MOV     CX，100
    L1：
        ADD     SI，2
        CMP     TABLE [SI]，－1
        LOOPE   L1
```

这三个循环指令摘要在表 9-7 中列出。注意，LOOPE 与 LOOPZ 是相同的，而 LOOPNE 与 LOOPNZ 是相同的。因为处理器在 CMP 指令通过检查 ZF 测试是否相等，因此如果比较相等，则 ZF＝1，否则 ZF＝0。

表 9-7 循 环 指 令

指　令	说　明
LOOP	自动减 1。当 CX＝0 时，循环终止。循环的顶端距离 LOOP 指令不可超过 128 bytes。（这适用于所有 LOOP 指令）
LOOPE/LOOPZ， LOOPNE/LOOPNZ	当相等或不相等时，执行循环。两个都会检查 CX 和 ZF。LOOPZ 是当 CX＝0 或 ZF＝0 时，终止执行。LOOPNZ 是当 CX＝0 或 ZF＝1 时，终止执行。 LOOPE 和 LOOPZ 编译成相同的机器指令，LOOPNE 和 LOOPNZ 也一样。依照你循环的内容选择较适合的。如果你不要使用计数器（CX）去控制循环，可以将 CX 设置成超出范围的数字。
JCXZ，JECXZ	如果 CX＝0 或 ECX＝0 则分支到一个标记。不像其它的条件转移，在 80386/486 可以跳至 NEAR（在目前段中，64KB）或 short（－128～＋127）的标记，JCXZ 和 JECXZ 只能跳至 short 的标记。

JCXZ 和 JECXZ 指令提供一个当循环执行时循环计数器 CX 是空的方法。例如考虑下列的循环：

```
mov   CX，LoopCount      ; Load loop counter
next：    .              ; Iterate loop CX times
          .
          .
          loop next      ; Do again
```

如果 LoopCount 是 0，则第一次 CX 减 1 后为－1。然后必须再减 65535 次才到达 0。使用 JCXZ 去避免此问题：

```
mov   CX，LoopCount      ; Load loop counter
      jcxz  done         ; Skip loop if count is 0
next：    .              ; else iterate loop CX times
          .
          .
          loop next      ; Do again
done：                   ; Continue after loop
```

9.7 IF 伪指令

你可以使用 .IF、.ELSEIF、.ELSE 去形成与高级语言一样的高级结构。这些伪指令可以产生条件转移。而接在 .IF 伪指令之后的运算式会被计算。如果是真，则下列的指令将被执行直到下一个 .ENDIF、.ELSE 或 .ELSEIF 伪指令到达为止。如果运算式是假，则 .ELSE 语句将被执行。也可以使用 .ELLSEIF 伪指令再放一个新的运算式在原来的 .IF 语句中。文法是：

```
.IF   condition1
      statements
[.ELSEIF condition2
      statements]
[.ELSE
      statements]
.ENDIF
```

此结构

```
.IF   CX==20   ；也可以加小括号，如 .IF (CX==20)
      mov dx, 20
.ELSE
      mov dx, 30
.ENDIF
```

产生程序码（在 DOS 下执行记得加/Sg 才可在列表文件中看到）：

```
                              .IF cx==20
0017  83 F9 14          *         cmp  cx, 014h
001A  75 05             *         jne  @C0001
001C  BA  0014                    mov  dx, 20
                              .ELSE
001F  EB  03            *         jmp  @C0003
0021                    * @C0001:
0021  BA  001E                        mov  dx, 30
                                  .ENDIF
0024                    * @C0003:
```

“==”与 C 一样，是等于的意思。在以前像下列的高级运算

```
IF (op1==op2)   THEN
    〈statement〉
ENDIF
```

在高级语言可以很简单地使用，不过在汇编你必须写成

```
            CMP   op1, op2
            JE    next _ label
            JMP   end _ if
next _ label:
            (statement)
end _ if:
```

是不是很麻烦。如果是复合式的 IF 使用，那就更加麻烦了。现在只要使用 .IF、.ELSE 或 .ELSEIF，就如在高级语言中使用一样方便。记得在最后加 .ENDIF 结束。

9.8 循环伪指令

MASM 6.1 提供了许多高级控制结构去为你产生循环结构。这些伪指令与 C 或 Pascal 的 while 和 repeat 循环相似。如果曾经学过 C 或 Pascal 的朋友简直如鱼得水，因为看来看去简直一样，只是换在汇编使用而已。这些伪指令可帮你设计程序时较容易又省时、易读。编译程序会为你产生适合的编译码。这些伪指令摘要在表 9-8。

表 9-8

伪指令 (Directive)	说　　明
.WHILE... .ENDW	语句在 .WHILE condition 和 .ENDW 之间，当条件 (condition) 是真则执行。
.REPEAT... .UNTIL .REPEAT... .UNTILCXZ	此循环至少执行一次，直到 .UNTIL 之后的条件是真才退出。会产生条件转移。
.BREAK	无条件退出 .REPEAT 或 .WHILE 循环。
.CONTINUE	无条件跳过剩余的语句到循环结束部分。

这些结构所做的就和在高级语言如 C 或 Pascal 中一样。有几点是要注意的：

这些伪指令会产生适当的处理器指令，并不会产生新的指令。他们需要有正确的说明是有符号数还是无符号数。

这些伪 指令会依照条件的结果为基准去执行相对应的指令。condition 可以是有符号或无符号数值计算的运算式。运算式中可使用如 C 的二元运算符 (&&, ||, 或!)，或状态标志。

编译程序需要知道在运算式的操作数是有符号或无符号数。记住有符号数数据说明时请使用 SBYTE、SWORD、SDWORD。

9.8.1 .WHILE 循环

和 C 或 Pascal 一样，.WHILE 循环的条件测试是在语句执行之前。当条件是真时才会重复执行循环里的语句。记得要使用 .ENDW 结尾。当条件变成假时，程序会执行 .ENDW 伪指令之后的第一个语句。.WHILE 伪指令会产生适当的比较和转移语句。文法是：

```
.WHILE  condition
```

语句

.ENDW

例如此循环会搬移 buf1 中的内容到另一个 buf2，直到'$'字符被发现：

```
.DATA
        buf1   BYTE   "This is a string",'$'
        buf2   BYTE   100 DUP (?)
.CODE
        sub    bx, bx          ; 目的是让 BX=0，当然也使用
                               ; 「 mov bx, 0」 或「 and bx, 0」
        .WHILE (buf1 [bx]! ='$')
                mov al, buf1 [bx]
                mov buf2 [bx], al
                inc  bx
        .ENDW
```

9.8.2 .REPEAT 循环

.REPEAT 伪指令所允许的循环结构与 C 的 do while 和 Pascal 的 REPEAT 循环相似。

循环会执行到 .UNTIL（或 .UNTILCXZ）之后的条件变成真才退出。有没有注意到条件总是在循环结束部分才被检查，所以循环至少会被执行一次。.UNTILE 伪指令会产生条件转移。语法是：

.REPEAT

语句

.UNTIL condition

.REPEAT

语句

.UNTILCXZ [condition]

condition（条件）也可以是 exp1==exp2 或 exp1!=exp2。当有两个条件被使用，第二个运算式（exp2）可以是一个立即运算式、寄存器或一个变量（如果 exp1 是一个寄存器）。

例如，下列的程序会将用户所键入的字符填入缓冲区（buffer）中。当 Enter 键（字符 13D=0Dh）被按循环结束：

```
.DATA
        buffer BYTE 100 DUP (?)
.CODE
        SUB bx, bx
        .REPEAT
                mov ah, 1
                int  21h                ; Get a key
                mov buffer [bx], al     ; Put it in the buffer
                inc  bx                 ; Increment the count
        .UNTIL (al==13)                 ; Continue until al is 13
```

.UNTIL 伪指令产生条件转移，但 .UNTILCXZ 伪指令产生一个 LOOP 指令，我们将以

列表文件中的实例说明。在列表文件中由编译程序产生的码前面都有一个“*”：

```
        ASSUME  bx: PTR SomeStruct

                        .REPEAT
                *@C0001:
                                inc     ax
                        .UNTIL  ax==6
                *               cmp   ax, 006h
                *               jne    @C0001
        .REPEAT
                *@C0003:
                                mov    ax, 1
                        .UNTILCXZ
                *               loop    @C0003

                        .REPEAT
                *@C0004:
                        .UNTILCXZ   [bx].field1 =6
                *               cmp     [bx].field, 006h
                *               loope   @C0004
```

由.UNTILCXZ 的 CXZ 大概可以猜到如果要用 LOOP 指令应该是用 LOOPE 或 LOOPNE。不过上例是当不等于 6（! =6）时，退出循环，所以要用 LOOPE。要看清楚。

9.8.3 .BREAK 和 .CONTINUE 伪指令

.BREAK 和 .CONTINUE 伪指令可先前终止 .REPEAT 或 .WHILE 循环。这些伪指令允许一个选择性的 .IF 去造成条件循环中断。语法是：

.BREAK [.IF condition]

.CONTINUE [.IF condition]

注意在 .BREAK 和 .CONTINUE 中使用 .IF 不需要 .ENDIF 来结尾。.BREAK 和 .CONTINUE 与 C 的 break 和 continue 指令一样。

.BREAK 会无条件退出自己所处的最内层循环，直接跳开最内一层（最接近自己的一层）的循环。.CONTINUE 会直接跳至与它最接近的循环条件判断地方。如果是网状，有几个循环就要用几个 .BREAK 或 .CONTINUE，才能退出所有的循环。若不清楚请参阅任何 C 语言的有关介绍。

下列的循环只接受范围在 ‘0’ ～ ‘9’ 之间的按键，直到按 Enter 退出。注意 WHILE1 是死循环，因为 1 是真。

```
.WHILE 1                        ; Loop forever
    mov ah, 08h                 ; Get key without echo
    int  21h
    .BREAK .IF al==13           ; If Enter, break out of the loop
```

```
        .CONTINUE.IF (al<'0') || (al>'9')
                                ; If not a digit, continue looping
        mov dl, al              ; Save the character for processing
        mov ah, 02h             ; Output the character
        int 21h
    .ENDW
```

如果你在 DOS 命令行下编译先前的程序使用/F1 和/Sg 选项（/F 表示要产生列表文件；LST 文件。/Sg 表示将编译程序替我们所产生的程序码也列出在 LST 文件，详细的使用请参阅附录），在列表文件中你将看到下面的内容：

```
                        .WHILE  1
0017                 * @C0001:
0017  B4  08                    mov ah, 08h
0019  CD  21                    int  21h
                                .BREAK  .IF  al==13
001B  3C  0D         *                  cmp  al, 00DH
001D  74  10         *                  je   @C0002
                                .CONTINUE  .IF (al<'0') || (al>'9')
001F  3C  30         *                  cmp  al,'0'
0021  72  F4         *                  jb   @C0001
0023  3C  39         *                  cmp  al,'9'
0025  77  F0         *                  ja   @C0001
0027  8A  D0                     mov    dl, al
0029  B4  02                     mov    ah, 02h
002B  CD  21                     int     21h
                            .ENDW
002D  EB  E8         *           jmp    @C0001
002F                 * @C0002:
```

这些高级控制结构可以网状化。也就是 .REPEAT 里可包含 .REPEAT 或 .WHILE，.WHILE 里可包含 .REPEAT 或 .WHILE，当然 .IF 可以使用在其中。

9.9 编写循环条件

你可以使用相关运算符去表示 .IF、.REPEAT、.WHILE 伪指令的条件。你也可以使用 PTR 运算符去表达操作数的属性。为了编写循环条件，你也需要知道编译程序如何去计算在条件运算式中的运算符和操作数。

9.9.1 运算式运算符

在 MASM 6.11 中的二元关系运算符（有两个操作数）与 C 的一样。这些运算符（表 9-9）将产生比较、测试、条件转移指令。

表 9-9 运 算 符

运算符	意 义
= =	Equal（相等）
! =	Not Equal（不相等）
>	Greater than（大于）
>=	Greater than or equal to（大于或等于）
<	Less than（小于）
<=	Less than equal to（小于或等于）
&	Bit test（TEST）
!	Logical NOT
&&	Logical AND
||	Logical OR

注意，.WHILE（X）和.WHILE（X! =0）是一样的。.WHILE（! X）和.WHILE（X==0）相同。学过 C 语言的朋友应该能领会它的涵义。

你也可以在高级控制结构的条件运算式使用标志名称作为操作数。

ZERO? CARRY? OVERFLOW? SIGN? PARITY?

例如，在.WHILE（CARRY?），进位标志的值决定了条件运算的结果。

9.9.2 有符号与无符号操作数

缺省值，运算式中的运算符运算的结果都是产生无符号转移（也就是都是正的，只往后转移）。然而，如果其中有一个运算的结果有符号，编译程序会考虑将整个运算视为有符号运算。

你可以使用 PTR 运算符去告诉编译程序在寄存器中有一个特别的操作数或是一个有符号的常数，如下：

```
.WHILE  SWORD  PTR  [bx] <=0
.IF          SWORD  PTR  mem1  > 0
```

若没有 PTR 运算符，编译程序会将 BX 的内容视为一个无符号值。看起来 PTR 好像无所不在。你也可以在.IF，.WHILE，.REPEAT 中使用 SBYTE、SWORD、SDWORD 去指定操作数的大小属性：

```
        .DATA
mem1  SBYTE  ?
mem2  WORD   ?
      .IF        mem1>0
      .WHILE  mem2<bx
      .WHILE  SWORD  PTR  ax < count
```

和 C 一样，你可以使用下列的运算符去连结两个条件运算——&&（AND）、||（OR）、!（NOT）。这些运算符优先权为! → (&&, ||)。! 有最高的优先权。就像在大部分高级语言一

样，是由左往右计算。

9.9.3 条件运算式的计算方式

编译程序在计算由高级控制结构所建立的条件运算式时，是采用一种叫“short-circuit”的方式计算。如果有一个特别的条件计算的结果自动决定了最后的结果（比如有一个条件：使用 AND 连接的复合语句，计算的结果是假），其余的计算将不会再继续。

例如，在 .WHILE 语句中：

```
.WHILE (ax>0) && (WORD PTR [bx] ==0)
```

编译程序会先计算第一个条件，如果是假（也就是如果 AX 小于或等于 0），此计算就已完成，也就是第二个条件便不会去检查了，而循环也不会执行。这是因为此复合条件包含了 && 运算符（需要两个运算式都是真，整个条件才是真）。

9.10 字符串处理

8086 家族指令集有七个字符串指令，可快速且有效地处理全部的字符串和数组。“在字符串指令”里的“字符串”名词，指的是一连串的元素，不是单指字符字符串。在 8086～80286 处理器这些指令只可直接处理字节字数组，在 80386/486 上可处理 bytes、words、doublewords。对于要处理更大的元素必须间接使用循环处理。表 9-10 中列出本章节要说明的五个指令：

表 9-10 字 符 串 指 令

指 令	说 明
MOVS	从一个位置或另一个位置拷贝一个字符串。
STOS	存储累加寄存器（EAX）的内容到一个字符串。
CMPS	比较两个字符串。
LODS	从一个字符串载入一个值到累加寄存器中。
SCAS	扫描一个字符串一个指定值。

CX，100。如果你希望字符串指令在某一条件下被终止（例如，在搜寻的过程中当有一个符号被发现时），可通过载入最大的重复次数，避免在执行时发生错误。

（3）载入源字符串的起始地址到 DS：SI，和目的字符串的起始地址到 ES：DI。有些字符串指令只须一个目的或源字符串。正常源字符串的段地址应该是 DS，但是对于源操作数，你也可以使用段强定去指定不同的段。对于目的字符串你不可以使用段强定。因此你可能需要去改变 ES 的值。

虽然你可以使用段强定在源操作数，但是强定段结合重复前置指令，在所有的处理器（除了 80386/486），有某些情况可能造成困难。例如，在字符串运算的过程中，如果有中断发生，这段强定将失去功用，而其余的字符串运算处理将不正确。如果使用 80386/486 处理器或当中断被关掉时，段强定才可被安全使用。

任何特别字符串运算处理，你可以采用下列的步骤。语法如下：

```
[prefix]    CMPS    [segmentregister:]    source, [ES:] destination
            LODS    [segmentregister:]    source
[prefix]    MOVS    [ES:]    destination, [segmentregister:]    source
[prefix]    SCAS    [ES:]    destination
[prefix]    STOS    [ES:]    destination
```

许多指令对于 byte、word 或 doubleword 操作数有特别的形式。如果你在使用 LODS、SCAS 和 STOS 时，在结尾加上 B（BYTE）、W（WORD）或 D（DWORD），编译程序知道是在 AL、AX 或 EAX 寄存器的元素。因此这些指令的型式是不需要操作数的。

表 9-11 列出每个字符串指令使用重复前置指令的形式和是工作在源、目的操作数或两者。

表 9-11　字符串指令的需要

指　令	重复前置子	源/目的	寄存器组
MOVS	REP	两者	DS：SI，ES：DI
SCAS	REPE/REPNE	目的	ES：DI
CMPS	REPE/REPNE	两者	DS：SI，ES：DI
LODS	没有	源	DS：SI
STOS	REP	目的	ES：DI
INS	REP	目的	ES：DI
OUTS	REP	源	DS：SI

这些重复前置指令造成的重复执行（重复的次数指定在计数寄存器 CX），或直到条件变真。每次执行之后，会自动增加或减少 SI 或 DI，以使能指到下一个数组元素。由方向标志会决定 SI 和 DI 是要累加（标志清除）或累减（设置标志）。指令的大小会决定 SI 和 DI 每次是要改变 1、2 或 4。

每个前置指令重复的次数如下：

前置子（Prefix）	说明
REP	重复指令 CX 次。

REPE，REPZ　　当值（源与目的操作数）相等的重复指令最大 CX 次

REPNE，REPNZ　　当值不相等时重复指令最大 CX 次。

前置指令一次只能应用一个字符串指令。若要重要一个区段的指令，可使用循环指令。在程序执行时如果字符串指令之前有重复前置指令，处理器：

（1）检查 CX 寄存器，如果 CX 是零则退出。

（2）执行字符串运算一次。

（3）如果方向标志是清除则累加 SI 或 DI。如果方向标志是设置则累减 SI 或 DI。累加或累减的总数对于位组（byte）运算是 1，对于字组（word）运算是 2，对于双字组（doubleword）运算是 4。

（4）递减 CX（减 1），且不修改方向标志。

（5）如果 REPE 或 REPNE 前置指令被使用，检查 ZF（SCAS 或 CMPS）。如果重复的条件存在，循环跳回至步骤（1）。否则循环结束，执行下一个指令。

当循环结束时，SI（或 DI）会指到下一个符号的位置（使用 CMPS 或 SCAS），所以你必须增加或减少 SI 或 DI 去指到最后符合且正确的位置。

虽然字符串指令（除了 LODS）通常都使用重复前置指令，但是也可以单独使用。在这些情况中，SI 和（或 DI）寄存器会根据指定的方向标志和操作数的大小来调整。

9.10.2　使用字符串指令

在这章节中你可以使用结构或联合的技巧，因为字符串或数组可以是结构或联合的区段。

1. 搬移数组数据

MOVS 指令会从一个内存区段拷贝数据至另一个区段。为搬移数据，首先需载入计数、源和目的地址至适当的寄存器，然后使用 REP 和 MOVS 指令。

```
.386
.MODEL  small
.DATA
        source BYTE 10  DUP ('012356789')
        destin BYTE 100 DUP (?)
.CODE
        mov ax, @data                   ; Load same segment
        mov ds, ax                      ;   To both DS
        mov es, ax                      ;   and ES
        ...
        cld                             ; work upwqrd
        mov cx, LENGTHOF source         ; Set iteration count to 100
        mov si, OFFSET source           ; Load address of source
        mov di, OFFSET destin           ; Load address of destination
        rep movsb                       ; Move 100 bytes
```

2. 填充数组

STOS 指令可存储某一个指定的值至字符串的每个位置。此字符串是指目的操作数，所以必须被 ES：DI 所指到，而指定的值必需载入累加寄存器中（AX）。

下面的实例会存储字符'a'在 100－byte 长字符串的每个位组（byte）中，也就是用'aaaaa.....'填充整个字符串。注意此程序是存储 50 words 而不是 100 bytes，这使得填充运算较快

(借助重复的次数)。

```
.MODEL  small
.DATA
        destin BYTE 100 DUP (?)
        ldestin EQU (LENGTHOF destin) /2
.CODE                                   ; Assume ES=DS
    ...
    cld                                 ; Work upward
    mov ax,'aa'                         ; Load character to fill
    mov cx, ldestin                     ; Load length of string
    mov di, OFFSET destin               ; Load address of destination
    rep  stosw                          ; Store 'aa'into array
```

3. 比较数组

CMPS 指令比较两个字符串，并指到下一个地址(符合或不符合)。如果是相等，ZF 被设置。无论哪一个字符串都可作为目的或源操作数，除非使用段强定。

下面的实例使用CMPS,假设字符串是在不同的段,两个段都必须被初始化至适当的段寄存器。

```
.MODEL  large
.DATA
        stringl BYTE "The quick brown fox jumps over the lazy dog"
.FARDATA
        string2 BYTE "The quick brown dog jumps over the lazy fox"
        lstring EQU LENGTHOF string2
.CODE
        mov ax, @data                   ; Load data segment
        mov ds, ax                      ; into DS
        mov ax, @fardata                ; Load far data segment
        mov es, ax                      ; into ES
        ...
        cld                             ; Work upward
        mov cx, lstring                 ; Load length of string
        mov si, OFFSET string1          ; Load offset of string1
        mov di, OFFSET string2          ; Load offset of string2
        repe cmpsb                      ; Compare
        je allmatch                     ; Jump if all match
        ...
allmatch:                               ; Special case of all match
```

4. 从数组载入数据

LODS 指令从字符串载入一个值到累加寄存器中（AX)。这个指令不能使用重复前置指令，因为连续载入至累加寄存器是没有什么意义的，因为只有最后一个值才会保留下来。

在下面实例中的程序片断会载入、处理、显示字符串中的每个 byte。

```
.DATA
```

```
        info BYTE 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        linfo WORD LENGTHOF info
.CODE
.
.
.
        cld                             ; Work upward
        mov cx, linfo                   ; Load length
        mov si, OFFSET info             ; Load offset of source
        mov ah, 2                       ; Display character funcution

get:
        lodsb                           ; Get a character
        add  al,'0'                     ; Convert to ASCII
        mov dl, al                      ; Move to DL
        int  21h                        ; Call DOS to display character
        loop get                        ; Repeat
```

5. 搜寻数组

SCAS 指令会比较借助 ES：DI 所指到的值和在累加寄存器（AX）中的值。如果两个值相等，设置 ZF 标志。重复前置指令可使 SCAS 指令处理整个字符串，也就是可扫描一个特别值称为目的操作数。REPNE SCAS 设置 ZF（如果发现这个目的值在数组中）。REPE SCAS 设置 ZF（如果被扫描的数组只有目的值）。

这个实例假设 ES 与 DS 不相同，而字符串的地址存储在一个指针变量。LES 指令会载入字符串的远程地址至 ES：DI。

```
        PBYTE  TYPEDEF  PTR  BYTE
.DATA
    string BYTE "The quick brown fox jumps over the lazy dog"
    pstring PBYTE string                ; Far pointer to string
    lstring EQU LENGTHOF string         ; Length of string
.CODE
.
.
.
    cld                                 ; Work upward
    mov  cx, lstring                    ; Load length of string
    les  di, pstring                    ; Load address of string
    mov  al,'Z'                         ; Load character to find
    repne scasb                         ; Search
    jne  notfound                       ; Jump if not found
    .                                   ; ES: DI pointers to character
    .                                   ;   after first 'Z'
    .
```

```
notfound:                          ; Special case for not found
```

6. 在字节数组中翻译数据

XLAT (Translate) 指令会从字节大小的数组中 copy 一个 byte 到 AL 寄存器。例如，给一个数字 7，XLAT 会回传数组中第 7 个字节。此数组可以存储 byte 大小的整数，通常是一个表格或一连串字符。语法：

XLAT [B]　　[[segment:] memory]

这字尾 B (byte) 是选择性的，反应了指令所处理的数据大小。XLAT 和 XLATB 事实上编译成相同的机器码。

为了使用 XLATB，需放置数组的开头偏移至 BX 寄存器中和欲求的索引值在 AL。在汇编语言数组的索引总是从 0 开始。如果要查询数组的第一个 BYTE，需设置 AL 为 0，如果要查询数组的第二个 BYTE，需设置 AL 为 1，以此类推。XLAT 会传回此欲求的字节元素在 AL，因此会覆盖掉原先在 AL 的索引值。

缺省值，DS 寄存器包含此表格的段，但你也可以使用段强定去指定一个不同的段。你不需要去给定一个操作数，除非当你使用段强定才需要。

下面的实例示范使用 XLAT 寻找在表格中的十六进制字符。这段程序会转换一个 8 bit 的二进制数字到一个以十六进制表示的字符串。

```
        ; Table of hexadecimal digits
          hex     BYTE  "0123456789ABCDEF"
          convert  BYTE  "You npressed the key with ASCII code"
          key     BYTE ?,?,"h", 13, 10,"$"
.CODE
.
.
mov ah, 8                          ; Get a key in AL
int  21h                           ; Call DOS
mov bx, OFFSET hex                 ; Load table address
mov ah, al                         ; Save a copy in high byte
and  al, 00001111B                 ; Mask out top character
xlat                               ; Translate
mov key [1], al                    ; Store the character
mov cl, 12                         ; Load shift count
shr ax, cl                         ; Shift high char into position
xlat                               ; Translate
mov key, al                        ; Store the character
mov dx, OFFSET convert             ; Load message
mov ah, 9                          ; Display character
int  21h                           ; Call DOS
```

虽然 AL 不能包含超过 255 的索引值，但你使用 XLAT 去使用超过 256 个元素的数组。只需将每 256 bytes 视为一个，就如同是一个子数组一般。例如，要查询数组第 260 个元素，

你可以加 256 至 BX（假设已存储了 hex 的偏移），再将 AL 设为 3（260－256－1）即可。之所以为 3 是因为数组索引值由 0 开始计算起。

下面 XLAT.ASM 程序可由键盘输入字符，因为是使用 DOSINT 21H 的第八号功能，输入的字符会存储在 AL，且没有回送至屏幕，而只有 32～127 之间的 ASCII 字符会被输出至屏幕。

```
 title Character Filtering Example    (XLAT.ASM)
.dosseg
.286
.model small
.stack 1024
.data
     table  BYTE  32  dup (0)
            BYTE  96  dup (0FFh)
            BYTE  128 dup (0)
.code
     main proc
          .STARTUP
            mov  bx, offset table
   getchar:
            mov  ah, 8
            int  21h
            cmp  al, 13
            je   exit
            mov  dl, al
            xlat table
            or   al, al
            jz   getchar
            mov  ah, 2
            int  21h
            jmp  getchar
      exit:
               .EXIT
   main endp
end
```

第 10 章　磁盘操作与文件处理

写完程序后，最重要的一件工作就是将程序 Save 起来。程序执行的结果或每次执行的进度，都需一个存储设备。你可能需要一个永久性的存储设备，将您重要的客户交易数据记录起来。存储设备一般有纸、RAM、磁带、磁盘……。存储在纸上将带来许多不便，那您买计算机做什么。存储在 RAM 上，姑且不论 RAM 的昂贵，那您可得保证永不断电，好像有点不太可能。如果您带着磁带去操作，可能会很累，还得提防被人笑。我们讨论的重点将在磁盘上。程序员最关心的是数据在磁盘上的放置及管理方式上。

10.1　数据磁盘对映

软盘（表 10-1）内部有一个磁性介质的圆型塑胶盘。磁盘双面都有磁性介质，因此双面都可存储数据。利用电机将磁盘做等速度旋转，读写头可任意移至磁盘上任一位置，可以磁化磁盘上磁性介质来存储数据。而且也能解读出原先的数据来。硬盘的结构与软盘相似，只是通常有很多圆盘及读写头。两种磁盘的圆盘每一面都有一个读写头，将代表数字化数据的磁性信号记录在磁盘上。

表 10-1　常见软盘格式

磁　盘	容　量	磁　面	磁道数	扇区数（每道）	磁　头
5 1/4 寸软式磁盘	360KB	2	40	9	2
	1.2MB	2	80	15	2
3 1/2 寸软式磁盘	720KB	2	80	9	2
	1.44KB	2	80	15	2

磁盘可分成磁道（Track）、扇区（Sector），柱面（Cylinder）。

磁道：将读写头固定，磁盘绕着读写头旋转一圈所形成的同心圆称为磁道。

扇区：每一磁道又可分为数个扇区，原则上每一扇区为 512 bytes 大小。相对于每一磁道和扇区都有连续号码来标定。因此只要指明磁道和扇区号码，便可找到磁盘上特定数据。

柱面：相对于每个读写头位置的一组同心圆称之为柱面。而且也跟磁道一样依序编号。一般是用在硬盘上。由于硬盘的密度和精确度都较高，所以相对地磁轨数和磁道的扇区数都较多。若要指定（读取）特定磁道时，需指明柱面号码及读写头号码。

在标准的 DOS 磁盘格式中，DOS 系根据“媒体描述字节”（media descriptor byte；表 10-2）的值来辨认各种磁盘格式。

表 10-2　媒体描述字节

磁盘容量	媒体描述字节	磁盘容量	媒体描述字节
5 1/4 寸软盘	360KB　FDH	3 1/2 寸软盘	720KB　F9H
	1.2MB　F9H		1.44MB　F8H

软硬盘对于数据的安排方式并没有不同，都是利用磁道、磁面、扇区号码汇编的方式去找寻数据。由于硬盘容量较大，可以将部分存储空间划分给不同操作系统，而只保留一部分给 DOS 使用，所以硬盘的使用空间最多可被分割成四个逻辑“分区 (Partitions)”。各分区内的数据分开独立，而不能相互存取，它们各自拥有自己的启动扇区及操作系统。

PC 上的软盘和硬盘的第一个扇区（柱面 0、磁头 0、扇区 1）都被预留做为开机启动程序（由于一个扇区只有 512 bytes，可见程序并不能太大）的存储区。启动程序（Bootstrap program）的工作是将磁盘上另一个操作系统启始程序（Start-up program）拷贝至内存中。

在开机后，ROM BIOS 所做的最后一件事就是将启动扇区读入内存中，并检查该扇区 (512 bytes) 的最后两个字节是否为 55h 与 0AAh。若是则表示此扇区所存储的为启动程序并可用来开机。

操作系统的启始程序的长度及存放位置并没有限制，所以可用来启动不同的操作系统。

由于软盘上的第一个实际扇区内容为 boot 程序，但硬盘上的第一个实际扇区是一个 partition 扇区，内有 boot 程序及 1 至 4 栏的分区表 (partition table)。Partition table 共 64 bytes 长，所以每个分割为 16 bytes。分区表记载硬盘每个分区的所在位置，此外也记载哪一个分区为“可启动分区 (Bootable Partition)”。可启动分区中的第一个扇区即为该分区的启动扇区，称为“分割启动扇区 (Partition Boot Sector)”。

开机时，会由 ROM BIOS INT 19H 读取硬盘中第 0 柱面第 0 磁面第 1 扇区的 load-boot 程序 (硬盘启动程序)，再由此程序辨别哪一个分区是可启动分区 (即 boot 扇区)。若找到，则将 boot 扇区内容读入内存中，再将控制权转移给 boot 程序，由 boot 程序将操作系统读入内存中，并将控制权交给操作系统。

一个硬盘可以分成若干个分区 (partition) 使用。DOS 提供 FDISK 程序可以将硬盘分成基本 DOS 分区 (Primary DOS Partition) 与扩展 DOS 分区 (Extended DOS Partition)，而扩展分区又可分成若干个逻辑 DOS 分区 (Logical DOS Partition)。而其它的操作系统也可制造出非 DOS 分区，当然 DOS 是无法运用到非 DOS 分区的硬盘空间。

另外，计算机的开机启动系统不一定要设在第一个分区中，即启动分区 (active partition)，所以我们知道硬盘的第一个实际扇区并不存放 boot 程序，而是存放 Load-boot 程序及 Partition table，由 Load-boot 程序去辨别哪一个分区为启动分区，启动分区内并不一定是存放 DOS 的 boot 程序也有可能是其它操作系统，由此就可以选择由任一种操作系统方式开机。但若有两个实际硬盘，开机系统必须设在第一部实际硬盘中，除非是修改 BIOS 程序。存取硬盘的启动扇区需小心，非常小心。一般启动扇区只是提供给 ROM BIOS 的启动载入程序 (Bootstap Loader) 使用，若其中的数据被破坏或删除，可能会导致整个硬盘数据无法存取。小心！小心！

10.1.1 逻辑扇区

在 BIOS 下，它是依据柱面、磁面、扇区的编号来读写每一个“实际的”扇区，而 DOS 并不知道什么是柱面、磁面、扇区 。DOS 将磁盘视为由一序列的逻辑扇区组成。逻辑扇区的编号是依同一磁道上，一个磁面接下一磁面的顺序编排下去，而上一个磁道排完后，才排下一个磁道。而在 DOS 逻辑扇区下，它往往将 boot 程序所在的扇区当作最起始的扇区 (逻辑扇区 0)，而 BIOS 的扇区编号则从 1 编起。Partition 扇区为硬盘的第 0 柱面、第 0 磁面、第 1 磁面 (BIOS 观点)，DOS DE-

表 10-3 逻辑扇区

保留区
文件分配表 (FAT)
根目录区
文件区 文件和子目录

BUG 程序的“L”指令是用来读取磁盘逻辑扇区，所以只能读到 boot 程序，而读不到 partition 扇区。不过可用 BIOS INT 13H 读取第 0 柱面、第 0 磁面、第 1 扇区的 partition，而 DOS 是以 boot 程序所在的启动扇区来开机，而前一个扇区它就管不到了。可见 DOS 并不是整个实际硬盘都管得到的，如非 DOS 分区就管不到。

由于软盘没有分区的问题，所以软盘的第 0 磁道、第 0 磁面、第 1 扇区直接对映逻辑扇区编辑 0，也就是 boot 扇区。

DEBUG L〈开始地址〉〈驱动器编号〉〈扇区编号〉〈扇区个数〉。

使用 debug 命令“L0001”的意思是从 A 驱动器逻辑扇区编号 0 起，读取一个扇区（即 boot 扇区）数据放内存地址。

〈开始地址〉：是指读取的数据所要放入内存的起始地址。

〈驱动器编号〉：0＝A，1＝B，2＝C，3＝D…

〈扇区编号〉：指所要读取的逻辑扇区起始编号。

〈扇区个数〉：总共要读取的扇区总数。

所以利用 debug 命令 L 就可读取指定驱动器的逻辑扇区，尤其是 boot 扇区（逻辑扇区）。

“L0001”：读取 A 驱动器 boot 扇区。

“L0101”：读取 B 驱动器 boot 扇区。

“L0201”：读取 C 驱动器 boot 扇区。

⋮

下面我们示范读取 C 磁盘的第 0 扇区，并将其放入偏移 0 的地址，再利用 D（dump）命令列出读取的数据（您的也许会与我的不太一样）：

```
    -l 0 2 0 1
-d⓪————→打印地址 0 的数据
0BB4：0000 EB 3C 90 4D 53 44 4F 53-35 2E 30 00 02 10 01 00 . (.MSDOS5.0.....
0BB4：0010 02 00 02 00 00 F8 FC 00-3F 00 10 00 3F 00 00 00.....?...?..
0BB4：0020 D1 BB 0F 00 80 00 29 E2-0F 6A 17 43 48 45 4E 20.....)...j.CHF
0BB4：0030 59 55 20 20 20 20 46 41-54 31 36 20 20 20 FA 33 YU FAT16

    0BB4：0040 C0 8E D0 BC 00 7C 16 07-BB 78 00 36 C5 37 1E 56.....|...x.6.7.V
0BB4：0050 16 53 BF 3E 7C B9 0B 00-FC F3 A4 06 1F C6 45 FE .S.>|.........E.
0BB4：0060 0F 8B 0E 18 7C 88 4D F9-89 47 02 C7 07 3E 7C FB ....|.M..G...|.
0BB4：0070 CD 13 72 79 33 C0 39 06-13 7C 74 08 8B 0E 13 7C ..ry3.9..|t....|
-q
```

10.1.2 磁盘格式

DOS 在执行软盘或硬盘格式化后，会将之分成 4 个区段。保留区（Reserved Area）、文件分配表（File allocation Table；FAT）、根目录区（Root Area）以及文件区（File Area）。每个区域大小长度随磁盘格式而有所不同。

保留区（表 10-4）的长度可能是一个或数个扇区。如果包含启动扇区（逻辑扇区 0），那一定是第一个扇区。在启动扇区中有一张表，记录保留区的长度、FAT 表的长度及份数（一般为 2 份），以及根目录区中所有目录的总数。软盘不管是否能开机至少会有一个扇区当成保留区。

FAT 表紧接在保留区之后。记录所有磁盘文件在磁盘空间的使用情况。包含文件占用的空间及未被使用的空间，以及磁盘损坏而不能使用的空间。我们保持有两份FAT 表（一模一样的两份），以便万一有一个FAT 表受损，还保有另一份FAT 表。因为如果FAT 表损坏，那么整个文件或目录在磁盘的存储位置都将流失，而造成数据丢失的困扰。FAT 表的长度（大小）会随磁盘大小不同而有不同大小。通常硬盘比软盘有较大的 FAT 表。540MB 的硬盘的FAT 表长度，比 120MB 的硬盘较大。

表 10-4　软 盘 格 式

磁盘	容量	保留区（扇区）	FAT 表长度（扇区）	根目录区长度（扇区）
5 1/4in	360KB	1	4	7
	1.2MB	1	14	14
3 1/2in	720KB	1	6	7
	1.44MB	1	18	14

根目录区是一张表格，利用目录项（Directory Entry）识别磁盘中的各个文件或目录。每个目录项 32 bytes（记录着文件名称、文件长度、文件在磁盘上的位置）。当然根目录的长度大小也是随磁盘的格式大小而不同。

也就是和 FAT 表的情况一样，较大的磁盘就需要有长度较大的 FAT 表和根目录区记录磁盘的使用情况。硬盘当然得看它的分区大小，才能决定 FAT 表和根目录区的长度大小。较大的分区，就需要有较长的长度。

文件区可存放文件和子目录数据，也是数据在磁盘上真正存储的空间。它占据了绝大部分磁盘可用空间。文件或子目录所存储的空间都是以一个磁簇（clusters）为单位。如果一个磁簇放不下，可以再存放到另一个磁簇，直到存储完毕或磁盘已满为止。

所谓磁簇指的是连续的一个或数个扇区的汇编，且磁簇所含的扇区数一般是 2 的次方。DOS 磁盘（表 10-5）的磁簇大小也随磁盘格式不同而有异。

表 10-5　DOS 磁盘磁簇大小

磁盘	容量	磁簇大小（扇区数）
5 1/4in 软盘	360KB	2
	1.2MB	1
3 1/2in 软盘	720KB	2
	1.44MB	1
硬盘（目前）	340MB	16
	540MB	16
	850MB	16

10.2　磁盘逻辑结构

10.2.1　根目录区

根目录（表 10-6）是由一连串 32 bytes 大小的目录项所组成。在这 32 bytes（表 10-7）中记录了文件（子目录、磁盘标号）名称、长度，位于磁盘的起始磁簇编号，最后一次修改文件的日期及时间。

1. 偏移 00H～07H：文件名称

DOS 在存储文件名称时是以主文件名（最多 8 个字符）＋副文件名（最多 3 个字符）的

表 10-6　常见 DOS 磁盘格式的根目录区长度及个数

磁　　盘	容　　量	长　度（扇区）	目录项个数
5 1/4in 软盘	360KB	7	112
	1.2MB	14	224
3 1/2in 软盘	720KB	7	112
	1.44MB	14	224
硬盘（目前）	340MB	32	512
	540MB	32	512
	850MB	32	512

表 10-7　一个目录项（32 bytes）结构信息

偏　　移	意　　义	长　度（Bytes）	存储格式
00H	文件名称（主文件名）	8	ASCII 字符
08H	扩展名	3	ASCII 字符
0BH	属性	1	位编码
0CH	保留	10	未使用（全部为零）
16H	时间	2	字组编码
18H	日期	2	字组编码
1AH	起始磁簇编号	2	字组
1CH	文件大小	4	整数

方式来存储文件名。00H～07H 前 8 个 bytes 是以 ASCII 字符格式来记录文件名称。若不满 8 个字符，则主文件名的其余位置则以空字符（20H）填充。文件名称一律为大写且字符间不能夹杂空字符。因为如 del、copy 是无法辨认中间夹有空字符的文件名称。不过 DOS 服务程序却可以辨识这种格式的文件名，也就是如果你不想被别人以 del、copy 等命令所更改时，就可以使用有空字符的文件名。

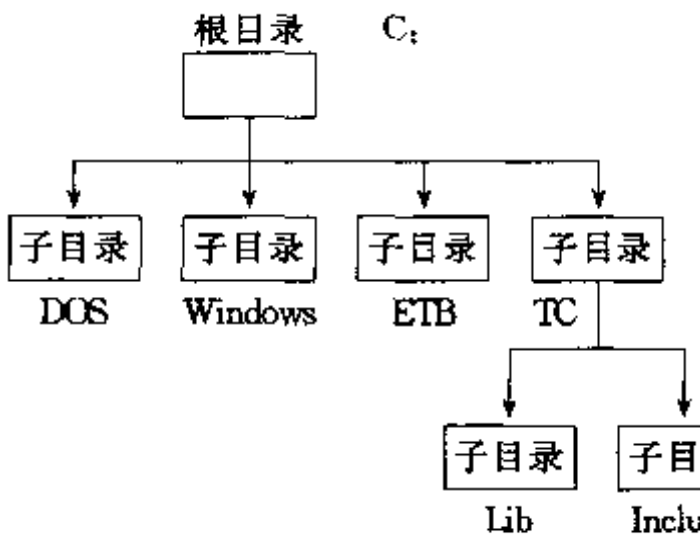
文件名称（表 10-8）的第一字节有一些特殊的使用情况。

表　10-8

第一个 byte 内容	代表的特殊意义
00H	表示此目录项从未被使用过。即整个磁盘所存储的目录项数据到此为止。即从这里以后都没有任何目录项存储了
05H	代表 05H 字符实际上是 E5H 字符（很少使用）
2EH	代表一个目录名称。起始磁簇会指向目录本身所在的磁簇中。若下一个 byte（即第 2 个 byte）也是 2EH 的话，则起始磁簇会指向其父目录所在的磁簇。若父目录为根目录时，第 2byte 则为 0
E5H	表示此文件已被记录为删除的文件或目录。可以用来再存储新的文件（即另一个文件名）

事实上当文件被删除掉（del）后，只有文件名称的第 1 byte 被改为 E5H。其余 31 bytes 的数据仍然保持着。如果这个目录项的位置没有被别的目录项所使用的话。尚可以把刚删除掉的数据救回来。如果被盖掉的话，那就回天乏术了。

目录的种类有两种：①一种是根目录（root directory）②另一种是子目录(subdirectory)。这种目录是用来存储磁盘中文件的名称位置。而根目录有固定的大小，并存储在固定的位置；子目录是附在根目录上，没有固定大小，没有固定位置。子目录与文件相同，只要还有磁盘空间的话，则子目录也可无限制的增加，再拥有其下的子目录，形成一个树状结构。如下图：



在根目录区中每一子目录与文件一样，都会有一个目录项，只是其属性标示其为一个子目录的目录项，且文件的大小设置为0。至于子目录下实际占用的空间大小可由FAT表中分配链获得。而一个子目录被建立后，用dir可以看到有两个特殊的项目置放其中，分别为“.”与“..”。“.”指子目录本身。例如可用“del.”删除子目录下所有文件。“..”指的是子目录本身的父目录。例如lib子目录的父目录为tc，而tc的父目录为根目录。我们可由根目录的起始磁簇编号为0，知道它为根目录。

如果要删除一个子目录，其下的所有文件若没有先被完全删除的话，则DOS是不会允许把子目录删除的。也就是曾经占据此空间的目录项虽已被消除，但其曾占用过而现在不再用的磁簇空间是不会空出的。

2. 偏移08H～0AH：扩展名

扩展名的长度为3个bytes，所以不满3个，一样是以空白字符（20H）填充。扩展名一般都定义一些较有意义的名字。如“.DOC”则较常代表文本文件。“.TXT”代表文字文件。

※　文件名不可以全部空白至少需有一个字符，但副扩展名却可以完全空白。

3. 偏移0BH：文件属性

文件属性占一个byte。(表10-9)，即每位各有其代表的意义。

表　10-9

7	6	5	4	3	2	1	0	位	意义
1	.	.	.	.	.	.	.		未用
.	1	.	.	.	.	.	.		未用
.	.	1	.	.	.	.	.		备存
.	.	.	1	.	.	.	.		子目录
.	.	.	.	1	.	.	.		磁盘标签
.	.	.	.	.	1	.	.		系统
.	.	.	.	.	.	1	.		隐藏
.	.	.	.	.	.	.	1		只读

位 0 若为 1，表示此文件为只读属性的文件，也就是无法删除或更改这个文件。有许多 DOS 服务程序会忽略此位，所以这不是绝对安全的方式（对保护文件数据不被删除或更改而言）。

位 1 和位 2 则用来标示文件为隐藏及系统属性。当文件被标示为隐藏或系统文件，或二种都标示之后，使用 dir 就看不到了。不过使用 dir/a 或 DOS 服务程序仍然可以查看标示“隐藏”或“系统”的文件。

位 3 则表示存储的为磁盘标签，所以只在根目录有意义。磁盘标签表示此磁盘的名称。在 DOS 命令行可用 Vol 来显示磁盘标签名。label 命令可用来更改磁盘标签名，不过只使用了目录项存储文件名的前 11 个 bytes，其余除了日期与时间区段仍记录外，均不使用。

位 5 为备存（archival）位。若位设为 1 表示此文件刚经过修改，还未备份。此位可协助磁盘作文件备份之用。若检查此位为 1，表此文件刚经过修改还未备份。若位为 0，表文件刚作过备份而尚未改变。

4. 偏移 0CH～15H：保留未用

共 10 bytes 的区段，暂时保留未用，以供未来发展之用。所有值均设为 0。

5. 偏移 16H～17H：时间

共 2 bytes 的区段用来表示此创建文件时间或最后一次被修改的时间。位 0～4H（共 5bits）：存储值范围可能为（0～29），以 2 秒为单位所以可以表示（0～58）秒。位 5～0AH（共 6bits）：存储值范围为（0～59），表示分钟。位 0B～0FH（共 5 bits）：存储值范围为（0～23）：用来表示小时。

1	0	0	1	0	0	1	1	1	1	0	0	0	1	0	1
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

（位）

（18：30：10）

6. 偏移 18H～19H：日期

共 2 bytes 用来表示此文件创建文件日期，或最后一次修改文件的日期。位 0～4H（共 5 bytes）：可能存储值范围（1～31），表示日期。位 5～8H（共 4 bytes）：可能存储值范围（1～12），表示月份。位 9～0FH（共 7 bytes）：可能存储值范围（0～127），表示年份。是以 1980 年为基址加上位 9～0Fh 所代表的二进制值为真正年份。

0	0	0	1	1	1	1	0	1	0	1	1	1	0	0	1
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

（1995 年 5 月 25 日）

※　虽然年份记录可达 2107（＝1980＋127）年，但 DOS 所能处理的最高年份为 2099 年。以目前来讲该是够用了。

7. 偏移 1AH～1BH：起始磁簇编号

共 2 bytes，用来表示文件存储在磁盘上的起始磁簇编号。我们可以利用这两个 byte 的值，到 FAT 表的文件分配链中去寻找文件的哪几个磁簇中。对于没有分配到空间的文件及磁盘标签名的目录项而言，起始磁簇的编号为 0。

8. 偏移 1CH～1FH：文件大小共 4 bytes，用来表示文件大小。最大可表示到 2^{32}bytes 的

文件大小。以目前的磁盘容量而言，已超过磁盘的最大容量。而 DOS 给定文件大小是以磁簇为单位，也就是 512 bytes 的倍数。所以文件数据真正的大小并不一定是这么大。也就是不一定是 512 bytes 的倍数这么刚刚好。

10.2.2 文件区

所有文件数据及子目录都是存放在文件区中，所以文件区是磁盘的最大部分。在理想的情况下，文件都是尽量存储在连续的空间中。当然也可以存放在不同的位置。当一个文件需要再扩充数据，或一个新文件要存储在已删除的文件空间上时，文件存放在不连续空间的情况甚为明显。

一般而言，将文件数据打散，存放在不连续的空间，将降低存取文件数据的速度。而且对一个已删除的文件要补救回来，会花较多的功夫来寻找该文件存储的扇区。

10.2.3 文件分配表（FAT）

文件分配表（FAT）是记录文件在磁盘空间的使用情况。虽然 DOS 会保持两份一模一样的 FAT，但在一般情况下，DOS 并不会去使用多余的那一份。在文件区中每一磁族在 FAT 表中都有一个项（Entry）与之对应。而在 FAT 表中每一项都会表示其为未使用、保留、坏的磁簇，或该 FAT 项所对映的磁族是否已有文件存放，且其 FAT 项的内容值指向文件存储的下一个磁簇编号。

一个文件所占用的磁盘空间，有时可能不是一个磁簇（一个或数个扇区）就能存储完毕，可能需要数个磁簇来存储，也就是需要数个 FAT 项来记录文件所占用的磁族编号。我们把数个 FAT 项的汇编称为一个链（chain）。可借助文件在根目录区的目录项中的启始磁簇编号找到该链的第一个磁簇编号，且磁簇编号在 FAT 表有一 FAT 项与之对应，而 FAT 项便是记录该链的第二个磁簇编号所对应的 FAT 项，这样循序下去直到文件结尾。

当有一个新的文件要建立或旧文件要再扩充数据时，DOS 会搜寻 FAT 表，找出还未使用的磁簇（FAT 项的内容为 0 的磁簇），并将该磁簇的空间配置给文件，然后将新增的磁簇加入该链中。

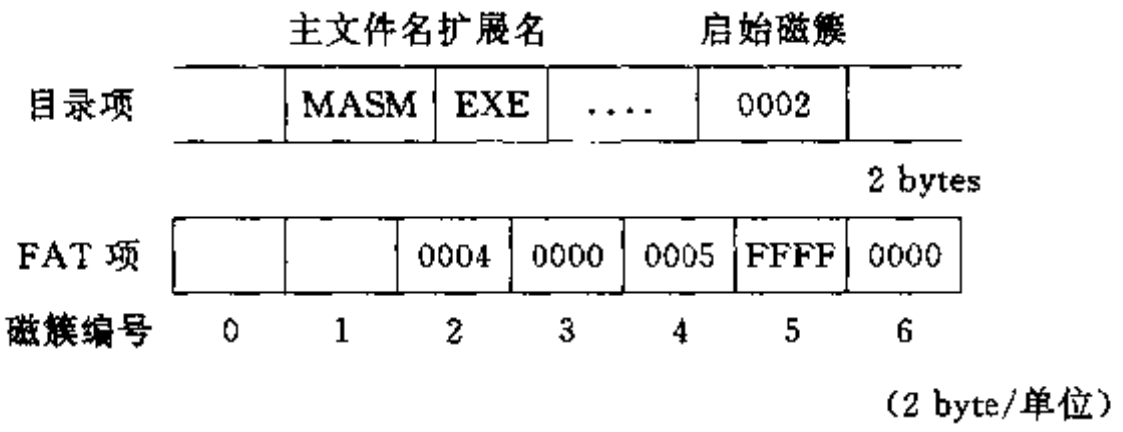
若有一个文件要缩减数据或要删除文件时，DOS 会清除所对应的 FAT 项，再将原文件所占用的磁盘空间(即所占用的磁簇编号）释放出来。

表 10-10　FAT 项的可能值

12 位值	16 位值	意　义
0	0	未用磁簇
FF0～FF6h	FFF0h～FFF6h	保留磁簇
FF7h	FFF7h	坏的磁簇
FF8h～FFFh	FFF8h～FFFFh	文件的最后磁簇
（其它值）		文件的下一个磁簇

FAT 项格式（表 10-10）可分为①12 位②16 位两种格式。

12 位的格式适用于软盘或硬盘总磁簇数未超过 4078 个。而 12 位的 FAT 项较难存取，原因是与 8086 微处理字库的 16 位字组不符。但 12 位格式的FAT项所占的磁盘空间较少。在 FAT 表中头两个 FAT 项是保留给 DOS 使用的。所以长度可能为 3 bytes（2 * 12 bits）或 4 bytes（2 * 16 bits）。而第一个 byte 是媒体描述值，与启动扇区中的 BIOS 参数集一样。而其余的 2 个或 3 个 bytes 则都填充 0FFh 值代表为保留磁簇。所以前两个磁簇编号 0 和 1 是保留，在文件区中文件真正存储的磁簇编号是从 2 开始编号，也就是我们文件数据是不可能存储在磁簇号码 0 和 1 的。

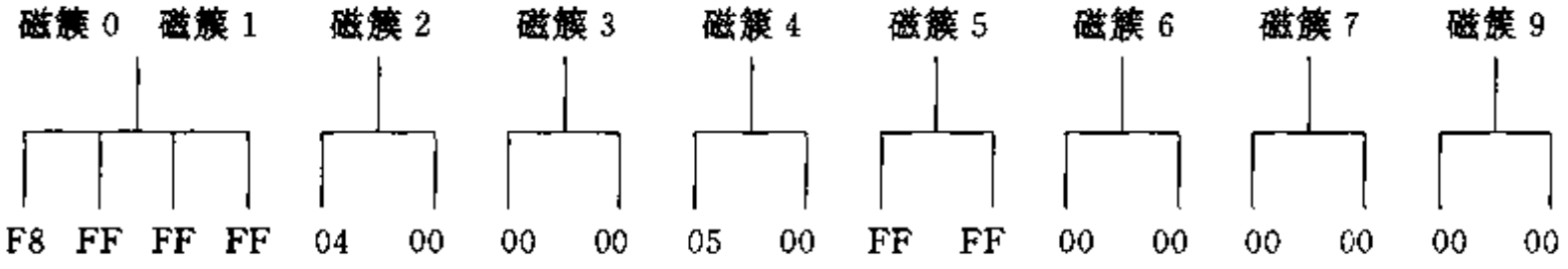


上图为 16 位格式的 FAT 项，记录一个文件名为 MASM. EXE 的配置情况。

读取 FAT 项方式

(1) 16-Bit FAT

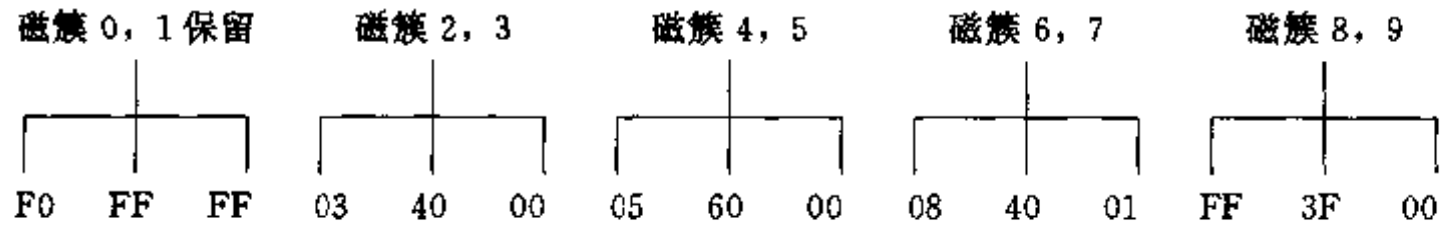
保留



读取 16 位格式 FAT 值只要将已知簇号码乘以 2，就可找到所对应 FAT 项的偏移量。例如磁簇 2 所对应的 FAT 项的偏移为 04H，其所对应的 FAT 项内容为 0004H，表示文件下一个磁簇编号为 4。所以所对应的 FAT 项为 4 * 2＝8H。又 08H 偏移的 FAT 项内容为 0005H，所以知道下一个磁簇编号为 5。又得到所对应的 FAT 项为 5 * 2＝10H，而 10H 偏移的 FAT 项内容为 0FFFFh。得知文件数据到此为止。

所以知道该文件存放在磁簇号码为 2，4，5 中。

(2) 12-Bit FAT



对 12 位格式的 FAT 而言，由于第二个 FAT 项共占 3 个 byte，所以要取得 FAT 项的偏移量，先将磁簇号码乘 3 再除 2 取得最后结果的整数值；(number * 3/2)。然后取出该偏移量所对应 FAT 项内容 (2 bytes)。而 2 个 byte 的十六进制有 4 个十六进制数。若先前的磁簇号为偶数，则舍弃最高位的十六进制数，若为奇数则舍弃最低位的十六进制数，剩下的三个十六进制数就是我们要的 FAT 项。

例如磁簇号码 2 的偏移量为 2 * 3/2＝3H。而偏移 03H 的内容为 4003H，因为磁簇 2 为偶数所以舍弃最高位 4 而得到 FAT 项为 003H，所以文件的下一个磁簇号码为 3。再把 3 * 3/2 得到偏移量 4H 的内容为 0040H，又因为上一个磁簇 3 为奇数，所以舍弃最低位，得到下一个磁簇号码 4H。再 4 * 3/2 得到偏移量 6H 的内容为 6005H，又上一个磁簇 4H，为偶数，所以舍弃最高位 6 得到下一个磁簇号码 5，再把 5 * 3/2 得到偏移量 7H 的内容为 0060H，又上一个磁簇 5 为奇数，所以舍弃最低位 0 得到一下一个磁簇号码 6。再把 6 * 3/2 得到偏移量 9H 的

内容为 4008H，又上一个磁簇 6 为偶数，所以舍弃最高位得到下一个磁簇号码 8H。再8 * 3/2 得到偏移量 12H 的内容为 3FFFh，又上一个磁簇 8 为偶数，所以舍弃最高位的 3 得到 0FFFh。文件存储到此为止。

所以我们可以知道该文件存放在号码为 2，3，4，5，6，8 的磁簇中。综上所述可得到下面法则：

①先将磁簇号码乘以 1.5，将得到的结果取整数，而得到在 FAT 表的偏移量 number。

②取出 FAT 表中偏移量 number 的内容（2 bytes），所以会有 4 个十六进制数。若上一个磁簇号码为偶数则舍弃最高位的十六进制数；若为奇数则舍弃最低位的十六进制数；所得到的值若不为 0FF8h～0FFFh（表最后一个磁簇），则为文件存储的磁簇号码。

③再重复①～②的步骤直到找到文件结尾为止。

※　这样找到的磁簇号码在 FAT 表中，就如同一串链一般。

不论是 12 位或 16 位格式的 FAT 表的前 2 个 FAT 项是保留而不是用来描述磁簇的使用情况，且最前面的第 1 个 byte 称为媒体描述字节（表 10-11）。若仅判别其值可能无法清楚辨别所有的软盘格式（因为有雷同）。

而 FAT 表的媒体描述字节虽可由 FAT 表最前一个 byte 获得，但一般我们还是利用 DOS 的服务程序 INT 21H，函数 1BH，来获得磁盘格式方法较佳。

表 10-11　软硬盘的媒体描述值

磁　　盘	容　　量	媒体描述值
5 1/4in 软盘	360KB	FDH
	1.2MB	F9H
3 1/2in 软盘	720KB	F9H
	1.44MB	F0H
硬盘		F8H

2. FAT 表的逻辑错误

FAT 表一般是由 DOS 所监管，但 FAT 表本身也可能会产生逻辑的错误。

①某个磁簇可能标示为使用中，但却不属于任何有效的分配链所有，变成孤儿。

②分配链上找不到文件结束的标示（0FFFh 或 0FFFFh）。

③任两个分配链同时占有一个磁簇，称为收敛（converge）。

④分配链可能发生循环现象而跳不出来，无法结束。

我们可以利用 DOS 的命令，chkdsk 和 recover 两个命令来检测与维护。除非有很好的理由，否则并不鼓励程序直接去存取磁盘的数据，如启动扇区数据以及 FAT 表和目录。

10.3　驱动器的管理

一般而言，磁盘的运作，最好由磁盘操作系统处理。还有就是执行、处理和监管磁盘输出、输入的运作，最好也都留给 DOS 处理，亦即通过程序语言所提供的文件服务程序，或 DOS 的服务程序。其中最主要的原因是因为让 DOS 处理会比较容易。DOS 有许多磁盘服务功能，如处理软盘所有的基本功能，包括格式化软盘，建立磁盘标签以及基本的磁盘读写操作。除了为了以绝对扇区方式处理磁盘数据需使用 ROM BIOS 服务程序外。

当我们查看所有程序语言、DOS、和 ROM BIOS 所提供的服务程序后，会发觉 DOS 所提供的磁盘服务程序最为丰富和方便。DOS 并不只是磁盘操作系统而已，而且是一个包含许多磁盘服务程序，可供程序使用的一个系统。事实上，我们可以发现 DOS（磁盘操作系统）是最适合管理磁盘运作的系统。

下面（表 10-12）我们介绍几个常用的 DOS 服务程序 INT 21H（有关驱动器管理功能）。

表 10-12　DOS 驱动器管理功能

AH＝0DH	磁盘重置	AH＝1BH	取得预设驱动器信息
AH＝0EH	驱动器选择	AH＝1CH	取得指定驱动器信息
AH＝19H	取得目前工作的驱动器代号	AH＝36H	取得磁盘剩余空间

10.3.1　0DH：磁盘重置

AH＝0DH，在有关文件的读写过程中，若执行此功能会把所有文件缓冲区数据回写至磁盘中，可更新文件的内容，但并不会自动关闭文件，也不会更新磁盘目录，也就是文件的实际大小和磁盘目录所记载的会有出入，若执行 AH＝10H（关闭文件）或 AH＝3EH（关闭文件代码），就可使文件大小和磁盘目录一致。

使用实例：

```
mov AH，0DH
int 21H；没有返回值
```

10.3.2　0EH：驱动器选择

AH＝0EH，将指定的驱动器代码设置在 DL，可以选择目前要使用的驱动器。会将已装设的驱动器数返回在 AL 寄存器。

驱动器代码 00 代表 A 驱动器，01 代表 B 驱动器，02 代表驱动器…。

若系统中的驱动器编号是利用 AL 值产生，传回值最小为 05H，即驱动器数若小于或等于 5，则返回值 AL 一定为 5，并不一定是真正的驱动器数。

使用实例：

```
mov AH，0EH
mov DL，02H      ；选择 C 磁盘
int 21H          ；回传磁盘总数在 AL
```

10.3.3　19H：取得当前所使用的驱动器代码

AH＝19H，此功能会将目前所使用的驱动器代码，存放到 AL 寄存器。DOS 是以标准数值代码（00H＝A 驱动器，01H＝B 驱动器，02H＝C 驱动器・・・）方式，将目前所使用的驱动器代码返回在 AL 寄存器中。

使用实例：

```
mov AH，19H
int 21H  ；回传所使用的驱动器代码在 AL
```

10.3.4　1BH：取得预设的驱动器信息

AH＝1BH，此功能可回传目前所使用的驱动器的相关数据，与 1CH 或 36H 服务程序类似。

回传参数	意　　义	回传参数	意　　义
AL	每个磁簇（cluster）的扇区数	DX	每个磁盘的总磁簇数
CX	每个扇区的长度（byte）	DS：BX	指向 FAT 表的 ID 字节

DS：BX 指向 FAT 表的 ID 字节，即指向 DOS 工作区的第一字节，此字节存放媒体描述值。由于此服务程序是利用 DS 寄存器传回媒体描述值。若您的程序是利用 DS 寄存器指向数据段（大部分的程序），所以为免 DS 寄存器被破坏，要先保存 DS 寄存器值。

使用实例：

```
push DS              ；保存 DS
mov  AH，1BH
int  21H             ；DS：BX→媒体描述字节
mov  AH，[BX]        ；先将媒体描述字节 copy 一份到 AH
pop  DS              ；回存 DS
```

10.3.5 1CH：取得指定驱动器信息

AH＝1CH，此功能和 1BH 服务程序一样，只是 1CH 功能可指定任一部驱动器，而 1BH 功能只能限定为当前所使用的驱动器。将指定的驱动器代号存入 DL 寄存器。

输入参数	意　　义
AH	1CH
DL	指定的驱动器代码（00H＝A，01H＝B…）
回传参数	**意　　义**
AL	每个磁簇的扇区数
CX	每个扇区的长度（字节数，bytes）
BX	磁盘的总磁簇数
DX：BX	FAT 表 ID 的地址

使用实例：

```
push DS
mov AH，1CH
mov DL，01H　；选择 B 驱动器
int 21H
mov AH，[BX]
pop DS
```

10.3.6 36H：取得磁盘剩余空间

AH＝36H，此功能和 1BH 与 1CH 功能类似，且会获得剩余的磁簇数。DL＝00H 为预设驱动器，即当前所使用的驱动器。

DL＝01H：表 A 驱动器；DL＝02H：表 B 驱动器；DL＝03H：表 C 驱动器…。

输入参数	意　　义
AH	36H
DL	驱动器代码（如上）
回传参数	**意　　义**
AX	选定错误（如无效的驱动器），AX＝0FFFFh 正确，AX＝每个磁簇的扇区数
BX	剩余的磁簇数
CX	每个扇区的长度（bytes）
DX	磁盘的总磁簇数

可以依照上面数据计算下列空间：

AX * CX　每个磁簇的总字节

AX * BX * CX　剩余空间的字节

AX * DX * CX　整个磁盘的总存储空间（bytes）

(BX * 100) /DX　剩余空间占总磁盘空间的百分比

使用实例：

```
mov AH，36H
mov DL，00H　；取得当前使用的驱动器信息
int 21H
```

10.4　DOS 目录管理

当我们在调用 DOS 服务程序之后，大都具有两个重要特色。

①大部分的服务程序，都会回传一个标准错误码在 AX 寄存器。当有错误发生时，进位标志会被设为 1。可以在每次调用这些服务程序后随即检查进位标志 CF（用 JC 指令）。若 CF=1，则 AX 寄存器中存放的值，即代表错误的原因。

②若服务程序需要输入字符串时（如路径名称），都是使用 ASCII 字符串的格式。所谓 ASCII 字符串，实际上就是一串 ASCII 字符，且在结尾处加上一个空字符（0H）标示字符串的结束。

下面（表 10-13）我们介绍几个常用的 DOS 服务程序 INT 21H（有关目录管理的几个服务程序）。

表 10-13　目录管理的服务程序

AH=39H	建立子目录	AH=47H	获取当前 工作目录路径
AH=3AH	删除子目录	AH=4EH	寻找第一个符合条件的文件
AH=3BH	改变当前的工作目录	AH=4FH	寻找另一个符合条件的文件
AH=41H	删除文件	AH=1AH	设置磁盘传送地址
AH=43H	获取及设置文件属性		

10.4.1　39H：建立子目录

AH=39H，和 DOS 的指令 MKDIR（MD）一样，会依照 DS：DX 这个地址所表示的路径，建立一个子目录。路径规格是以 ASCII 字符串格式设置，也就是此 ASCII 字符串必需包含此新目录的路径名称。

若执行时发生错误，会设置进位标志（CF=1），及标准错误码在 AX。

使用实例：

```
mov AH，39H
mov DX，offset path
int 21H
jc error
```

```
    ...
error:
    ...
    .data
    path BYTE '\MASM', 0
```

有可能发生的错误：

①AH＝03H，找不到指定的路径。

②AX＝05H，被拒绝执行。（可能磁盘已满，或已有相同的目录名称）

10.4.2 3AH：删除子目录

AH＝3AH，和 DOS 的命令 RMDIR（RD）一样。此功能会删除 DS：DX 地址所表示路径的子目录。路径一样是以 ASCII 字符串格式设置。

若执行时发生错误，会设置进位标志（CF＝1）及通过 AX 传回标准错误码。

使用实例：

```
    mov AH, 3AH
    mov DX, offset path
    int 21H
    jc error
    ...
error:
    ...
    .data
    path BYTE 'C: \MASM', 0
```

有可能发生的错误：

①AX＝03H，找不到指定的路径。

②AX＝05H，拒绝执行。（可能子目录不是空的、欲删除根目录、或欲删除的不是子目录）。

③AX＝10H，意图删除使用中的目录。

10.4.3 3BH：改变当前的工作目录

AH＝3BH，和 DOS 的命令 CHDIR（CD）一样。可以改变当前使用的目录（工作目录）到 DS：DX 地址所表示的路径。路径规格必须是一个 ASCII 字符串。字符串长度限定在 64 字符之内。

若执行时发生错误，会设置进位标志（CF＝1），通过 AX 传回标准错误码。

使用实例：

```
    mov AH, 3BH
    mov DX, offset path
    int 21H
    jc error
    ...
error:
```

```
...
.data
path BYTE 'C：\MASM611\INIT'，0
```

可能发生的错误：

①AX＝03H，指定的路径无效。(可能是指定的路径不存在，或指定的路径是一个文件名，而不是目录。)

10.4.4 41H：删除文件

AH＝41H，此功能可以删除以 DS：DX 地址所表示路径的下一个文件（目录项）。路径是以 ASCII 字符串设置，其中必须包含路径名称及文件名称。

①此功能一次只能删除一个文件，不接受 DOS 通配符‘*’及‘?’等。

②“只读”属性的文件中此功能无法删除。须先调用 43H 服务程序，改掉文件的只读属性。

若执行时发生错误，会设置进位标志（CF＝1），及通过 AX 传回标准错误码。

使用实例：

```
    mov AH，41H
    mov DX，offset file-path
    int 21H
    jc error
    ...
error：
    ...
    .data
    file-path BYTE 'A：TEST.ASM'，0
```

可能发生的错误：

①AX＝02H，找不到路径名称或文件名称。

②AX＝05H，拒绝执行。(可能所设置路径的是目录或是只读属性。)

10.4.5 43H：获取或设置文件属性

AH＝43H，此功能可以获取或设置指定文件的属性。

若是文件属性的获得，则将 AL 寄存器设为 0。文件路径名称的起始地址设在 DS：DX。文件的属性会读入 CX 寄存器。

若是文件属性的设置，AL 寄存器设为 1、CX 寄存器设置文件的属性。

若执行错误时，会设置进位标志，且将错误码置于 AX 中。

使用实例：获取文件属性

```
    mov AH，43H
    mov AL，0
    mov DX，offset file-path
    int 21H  ；回传文件属性在 CX
    mov attrib，cx
.data
```

```
file-path BYTE 'C：\MASM\MASM.EXE'，0
attrib WORD?
```

设置文件属性（表 10-14）时，文件属性指定在 CX 寄存器。

表 10-14　可能的文件属性

文件属性　备存位（Archive bit）	off	on	文件属性　备存位（Archive bit）	off	on
Normally	00H	20H	Hidden，read-only	03H	23H
Read-only	01H	21H	System	04H	24H
Hidden	02H	22H	Hidden，sytem，read-only	07h	27H

使用实例：设置文件属性

```
    mov AH，43H
    mov AL，1
    mov CX，3   ；(011b) 隐藏、只读属性
    mov DX，file-path
    int 21H       ；执行成功，回传文件属性在 CX
    jc error
    …
error：
    …
.data
    file-path BYTE 'TEST.ASM'，0
```

可能发生的错误：

①AX＝01H，表示 AL 设置有错。(可能 AL 是设置 00H、01H 以外的值)。

②AX＝02H，找不到文件。

③AX＝03H，找不到指定的文件路径。

④AX＝05H，拒绝存取。(可能 CX 设置的属性有错)。

10.4.6　47H：获取当前的工作目录路径

AH＝47H，此功能可以获得 DL 寄存器所指定驱动器，当前工作目录的路径，并将之存放在 DS：SI 所指定的缓冲区地址。传回的 ASCII 字符串路径名称最长达 64 bytes 长。

此功能回传的路径名称并不会包括驱动器的 ID（如 A：或 C：），代表根目录的反斜线('\')。如果返回的路径刚好是一根目录时，则回传的 ASCII 字符串将会是一个只有零字符的空字符串。

若执行错误时，进位标志将设为 1（CF＝1）。错误码回传在 AX。

使用实例：

```
    mov AH，47H
    mov DL，0   ；00H＝当前工作驱动器，01H＝A 驱动器…
    mov SI，offset directory-path
    int 21H
    jc error
```

```
    ...
error:
    ...
.data
    directory-path BYTE 64 DUP (?)
```

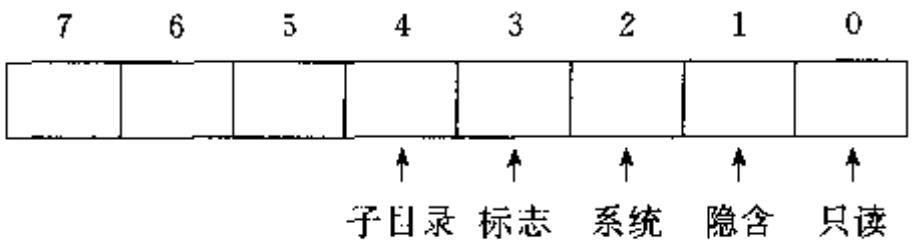
可能发生的错误：

AX＝15H，指定的驱动器不合法（不存在）。

10.4.7 4EH：寻找第一个符合条件的文件

AH＝4EH，此功能可用来寻找第一个符合条件的文件。必需先在 DS：DX 地址存放路径和文件名称的 ASCII 字符串。CX 寄存器设置寻找文件的属性。文件名称可以包含 DOS 的通配符'?'和'*'。

CX 寄存器属性可能的设置如下：



文件属性分别为：一般文件（00H），隐含文件（02H），系统文件（04H），子目录文件（10H）。若 CX＝（02H＋04H＋10H）＝16H，则可达到搜寻所有文件的功能。

若执行成功，会清除进位标志，再由 DTA（Disk transfer area）传回文件名称及相关数据（表 10-15）。

表 10-15 返回在 DTA 中的数据共 43 bytes

偏移（Hex）	长度（byte）	意　　义
00H	21	DOS 用在寻找下一个文件的区段
15H	1	文件的属性
16H	2	文件的时间
18H	2	文件的日期
1AH	4	文件大小（byte）
1EH	13	文件名称（文件名＋扩展名）ASCII 字符串

若执行错误，设置进位标志（CF＝1），传回错误码在 AX 中。

使用实例：

```
    mov AH, 4EH
    mov CX, 0   ; 一般文件（00H）
    mov DX, offset file-path
    int 21H
    jc error
    ...
```

```
error:
    ...
.data
    file-path BYTE '\MASM\*.ASM', 0
```

可能发生的错误：

①AX=02H，未发现文件。(可能文件不存在)

②AX=03H，找不到路径。

③AX=12H，没有其它相符的文件。

10.4.8 4FH：寻找另一个符合条件的文件

AH=4FH，此功能会寻找 4EH 或 4FH 服务程序所指定路径下一个符合条件的文件，也就是呼叫 4FH 功能时，必在之前呼叫过 4EH 或 4FH 的服务程序。DTA 中必须已存有先前呼叫 4EH 或 4FH 功能所传回的数据。之所以可能有多个符合条件的文件，是因为可能使用了通配符 '*' 或 '?'。

若找到符合条件的文件，会清除进位标志，并更新 DTA 中的内容。

若失败，则设定进位标志，并将错误码回传在 AX 中。

使用实例：

```
    mov AH, 4EH
    mov CX, 0
    mov DX, file-path
    int 21H
    jc error
repeat-read:
    mov AH, 4FH
    int 21H
    jnc repeat-read
    ...
error
    ...
.data
    file-path BYTE 'C:\MASM\*.ASM', 0
```

可能发生的错误：

①AH=12H，已无符合条件的文件。

10.4.9 1AH：设置磁盘传送地址 (DTA)

AH=1AH，此功能可建立 DOS 用来做文件 I/O 的磁盘传送区 (Disk Transfer Area：DTA)。我们须先指定 DTA 的地址在 DS：DX 这对寄存器上，然后才能使用 INT 21H 服务程序存取 DTA 的内容。

若我们不指定 DTA 的地址，DOS 会自动将 DTA 地址设置在 PSP 中偏移 80H 的地方。缺省 DTA 为 128 bytes (80H)。

使用实例：

```
        mov AH, 1AH
        mov DX, offset DTA-ADDRESS
        int 21H
        ...
.data
        DTA-ADDRESS BYTE 128 dup (0)
```

我们之所以要重改 DTA 地址有两个原因。

①因为缺省的 DTA 的位置的和存储 DOS 命令参数的位置是一样，也就是 DOS 在第一次去执行磁盘文件 I/O 后，就会破坏了 DOS 命令参数的数据。

②当一个 EXE 文件被载入内存执行时，DS 和 ES 是指向 PSP 的开头地址，而不是指向数据段。若我们程序中要使用数据段内的数据，势必要再重设 DS 指向数据段，从而失去 PSP 段的地址，而造成要做文件 I/O 存取数据时变得较困难。

如果是用 .STARTUP 伪指令开始程序，则 DS 与 ES 是指向数据段的。

10.4.10 PSP：（代码段前置区；Program Segment Prefix）

当 DOS 要载入一个程序执行时，它会在内存中保留一段给程序使用，称为程序段（program segment）或指令段（code segment）。在这个程序段前 100H（256 bytes）的字节称为 PSP（program segment prefix），而真正的程序通常是直接置在 PSP 之后。

PSP 中包含一些 DOS 用来执行一个程序所需的数据，不论何种语言的 DOS 程序都会有一个 PSP（包含 EXE 文件和 COM 文件）。一般高级语言程序设计是不会去理会 PSP 中的数据，因为高级语言在正常情况下都帮您处理妥当了。而汇编语言程序便很有可能去关心 PSP 中的数据。下面我们介绍 PSP 内部结构来说明程序与 PSP 之间的关系。

1. PSP 内部结构

偏移（Hex）	长度（byte）	意　　义
00H	2	INT 20H 指令码
02H	2	配置给程序的内存最后段地址
04H	1	保留
05H	5	DOS 服务程序分派 function 的远程调用地址
0AH	4	中断 22H 服务程序的地址
0EH	4	中断 23H 服务程序的地址
12H	4	中断 24H 服务程序的地址
16H	22	保留
2CH	2	环境区段地址
2EH	34	保留
50H	3	INT 21H 和 RETF 指令码
53H	9	保留
5CH	16	FCB1
6CH	20	FCB2
80H	128	命令行参数或缺省的 DTA 地址

00H～01H：存放 CDH 和 20H 两个 byte，也就是中断 20H 的指令码。中断 20H 是 DOS 用来结束程序执行的一种标准方法。只要把 CS 指向 PSP 的开头之处，IP＝0，程序就可跳到此处来结束执行。不过我们现在利用 DOS 21H，AH＝4CH 服务程序来结束程序的执行。

02H～03H：存放配置给程序执行的内存最后一个段地址。

05H～09H：这是使用 CP/M DOS 的调用地址，在 MS-DOS 我们直接利用 INT 21H 来调用 DOS 功能。

0AH～0DH：4 个 byte，DOS 中断 22H（结束程序）的地址。

0EH～12H：4 个 byte，DOS 中断 23H（Ctrl-C 中断处理）的地址。

12H～15H：4 个 byte，DOS 中断 24H（危急错误处理）的地址。

以上三个地址都是先存放 IP 低位地址再存放 CS 高位地址。

2CH～2DH：2 个 byte，存放由 DOS SET 或 PATH 命令所设置的环境变量的分段地址。环境区段是 ASCII 字符串所组成的串列，以长度为 0（空字符串）的字符串做为结束记号。例如 PATH＝C：\MASM。

50H～52H：3 个 byte，存放 INT 21H 指令码及 RETF（远程返回）。我们也可以利用此地址来调用 DOS 服务程序，但是要先载入输入参数在 AH，然后 CALL PSP＋50H。这种方式和直接使用 INT 21H 是一样的。这是另一种调用 DOS 功能的方法。

5CH～6BH：利用块式的文件处理。现在我们都使用新式文件代码（FILE HANDLER）来处理文件 I/O。

6CH～79H：同上。

80H～FFH：两种不同的用途：①可做为存储命令行参数。偏移 80H 记录参数字符串总长度，而 81H 才是真正记录参数内容的地址。②或做 DTA。

2. 命令行参数

在 DOS 下执行一个程序可能需要传一些参数给程序，如下：

read-str readme. DOC

假设有一个程序 read-str. EXE 需要传一个文件参数给程序，而“readme. DOC”就是我们所称的命令行参数。下面是命令参数的存储内容：

偏移	80H	81H	82H	83H	84H	85H	86H	87H	88H	89H	8AH	8BH	8CH
内容（Hex）	0B	20	52	45	41	44	4D	45	2E	44	4F	43	0D
			R	E	A	D	M	E	·	D	O	C	

第一字节（80H）记录命令行参数的字符总数。

当 DOS 在执行程序时会自动将命令行参数存放在 PSP＋80H 地址上，但并不会将启动程序的名称（即如下例 read-str. EXE）存放于此，而是从其后的第一个字符，也就是空白字符开始存储。而且命令行参数字符串之间的多余空白并不会自动删减，而是完完整整地储放。

在 C 语言的 main () 函数中有 argc 和 argv 来帮我们来取得命令行参数的个数和地址，而在汇编语言中就需要我们自己去取得。

下面实例说明如何取得命令行参数。如果顺利取得命令行参数，则会将之打印。这个程序有一点要注意，如果程序开头使用 .STARTUP 伪指令，是不能利用此方法来取得命令行参

数的。因为使用 .STARTUP 会将 DS 指向数据段（因为同在 DGROUP 群组中），而不是在 PSP 开头地址，所以需自己将数据段移入 DS。利用（MOV AX，@DATA）即可做到。注意 END 伪指令之后需指定程序的起始点。此例为 main。

```
.DOSSEG
.286
.MODEL SMALL
.STACK 1024
.DATA
        buffer byte 128 dup (0)
        end_ch byte '$'
        message byte 'No command tail','$'
.CODE
      main proc
            mov bx, ds          ; 先将 PSP 开头地址存储在 BX 中
            mov ax, @data       ; 需自己将数据段移入 DS 中
            mov ds, ax
            mov es, ax
            mov dx, offset buffer
            call get_com_tail
            jc no_input
            mov dx, offset buffer
            mov ah, 9
            int 21h
            jmp main_exit
no_input: mov dx, offset message
          mov ah, 9
          int 21h
main_exit: .EXIT
     main endp
     get_com_tail proc
          push ax
          push cx
          push si
          push di
          push es
               mov es, bx
               mov si, dx
               mov di, 81h
               mov cx, 0
               mov cl, es:[di-1]
               cmp cx, 0
               je L2
               mov al, 20h
               repz scasb
               jz L2
               dec di
               inc cx
```

```
L1: mov al, es: [di]
    mov [si], al
    inc si
    inc di
    loop L1
    clc
    jmp L3
L2: stc
L3: pop es
    pop di
    pop si
    pop cx
    pop ax
    ret
get _com _tail endp
END main
```

10.5 文件

DOS 提供十分完善的磁盘文件管理，由 DOS（Disk Operation System）的名称上可知是以文件为主的操作系统。本章将介绍 DOS 提供的文件代码和树状目录管理功能。

文件代码（file handle）是沿袭 Unix 系统。当要打开（open）或建立（create）一个文件时，DOS 会自动给此文件一个号码（代号），而一切的操作就只针对代号做处理，而不需直接对文件做处理。它的优点就是程序针对代号工作，而当要对不同的文件做处理时就不必更改程序中的文件。当文件关闭时，这个文件代号就必需交回给 DOS，以便下次可以再用。

在 DOS 的文件代号的文件管理概念里，会将常用的外围设备也看成文件来处理。当开机（系统启动）时，DOS 会自动设置 5 个特殊的文件代号（0、1、2、3、4），因此程序可以直接使用这些代号（表 10-16），而不必再 open 了。

表 10-16　5 个特殊的文件代号

Handle	外 围 设 备	备　　注
0	标准输入设备	常为 keyboard，可转向，仅能读取
1	标准输出设备	常为屏幕，可转向，仅能写入
2	标准示误设备	屏幕，不能转向，仅能写入
3	辅助串列 I/O 设备	I/O 可读写
4	标准打印设备	打印机，可读写

平时标准输出输入定为键盘和屏幕，若想要更改，可直接在 DOS 命令行下，以‘〈’或‘〉’做转向（redirection）即可。但标准示误设备不可转向。

C：\〉TYPE ASM.DOC〉LOOK.DOC

上例执行时会将标准输出（handle=1）设为 LOOK.DOC 文件而不再是屏幕。针对文件内数据的处理除了上一章介绍的，DOS 还提供了下列服务（表 10-17）：

表 10-17

AH	功　能	说　明
3CH	建立文件	建立一个新文件，或将已存在的文件的长度设为 0，以准备写入数据至此文件中，并传回文件 handle
3DH	打开文件	打开一个存在的文件，准备做输出输入数据。若成功则传回此文件代号
3EH	关闭文件	关闭一个文件代号。并将更改过后的数据写入文件并存入磁盘
3FH	读取数据	读取指定 handle 内的数据至一指定的缓冲区中
40H	写入数据	将数据写入指定的 handle 内
42H	移动文件指针	移动文件读写指针与指定的文件位置

10.5.1　3CH：建立文件及传回文件代号

AH=3CH，可建立一个新的文件或将一个已存在的文件长度设为零。3CH 服务程序并不会检查此文件是否已经存在。若建立的文件为一个已存在的文件，则会破坏此文件既存数据。

输入参数	意　义
AH	3CH
CX	文件属性
DX	DS：DX 指向所要建立的文件及路径名的 ASCII 字符串起始地址。
回传参数	**意　义**
CF=0	表示成功且 AX=文件代号
CF=1	表示失败且 AX=错误码

若打开的文件为第一次打开的文件，则 AX 通常为 5。

文件属性：

00H　正常文件
01H　只读文件
02H　隐藏文件
04H　系统文件

使用实例：

```
create _ file:
        mov AH, 3CH
        mov DX, offset file _ path
        mov CX, 0   ; 正常属性
        int 21H
        jc error
        mov file _ handle, AX
        ...
```

```
.DATA
file _ path byte 'ASM. DOC', 0
file _ handle word?
```

如果DS:DX路径所指定的文件为一个新文件(即文件不存在),则会将此文件长度设为零,若存在则将原文件数据清为零(即清除此文件内容),然后以“可读/可写”模式打开。

若要保护已存在的文件不被破坏,有两种方法。

①可先用3DH功能打开此文件,若此文件已存在则CF=1,传回错误码在AX。若AX=2表示文件没有发现,即不存在,你就可以安全使用此文件。

②可使用5BH服务程序打开文件,只是5BH会检查此文件是否已存在。若已存在则CF=1,AX传回50H表已存在。

使用实例①:

```
    mov AH, 3DH
    mov CX, 0
    mov DX, offset file _ path
    int 21H
    jc error
creat _ file:
    ...
```

使用实例②:

```
    mov AH, 5BH
    mov CX, 0
    mov DX, offset file _ path
    int 21H
    jc error
creat _ file:
    ...
```

可能发生的错误:

①AX=3,路径未发现。可能是DS:DX是一个不存在的路径名称。

②AX=4,打开太多的文件。在CONFIG.SYS中设置的files数目不够大,DOS缺省值为8。但系统一启动已打开了5个文件,只剩下3个文件可供打开使用。可将files数目设大一点如files=40,这样就还有35个可供程序利用。

③AX=5,拒绝存取。可能的情况是尝试建立一个已存在的文件且为只读属性,而不能打开来做输出之用;或是建立一个已存在的子目录;或磁盘已满无法再建立一个新文件。

10.5.2 3DH:打开文件

AH=3DH,此服务程序可用来打开一存在文件。打开的模式有只读、只写、可读写三种,指定在AL中。

7	6	5	4	3	2	1	0	位意义
1								继承标志
	S	S	S					共用模式
				R				保留未用
					A	A	A	读写存取

2	1	0	位意义
0	0	0	只读
0	0	1	读写
0	1	0	可读写

位 7 为继承位。若位 7 为 0 表子程序（process）可与父程序一样，使用同一文件代号存取该文件。若为 1 则子程序必须自行打开一文件，取得另一个文件代号。位 4～6 为共用模式位。

6	5	4	位	意义
0	0	0		兼容模式
0	0	1		拒绝读写模式
0	1	0		拒绝写入模式
0	1	1		拒绝读取模式
1	0	0		不拒绝模式

当尝试第二次打开同一文件时，DOS 会比较共用模式码及存取模式，码是否兼容，以决定是否接受第二次打开的要求。只有在网路上或已载入共用程序 SHARE，DOS 才会进行文件分享的操作。所以位 0～2 应是常注意的地方。平常位 3～7 设为零。

输入参数	意　义
AH＝3DH	
AL	打开模式
DX	指向文件路径名称的起始地址
回传参数	**意　义**
CF＝0	表成功，AX＝文件代码
CF＝1	表失败，AX＝错误码

使用实例：

```
mov AH, 3DH
mov AL, 0   ; 打开一个只读文件
MOV BX, offset file _ path
int 21H
jc open _ error
mov filehandle, ax
...
file _ path byte 'asm. doc', 0
filehandle word?
```

可能发生的错误：

①AX＝1，无效的功能号码。可能是程序尝试分享同一文件，而没有载入文件分享共用程序。

②AX＝2，文件没有发现。可能是指定的文件名称不存在。

③AX＝3，路径没有发现。表示指定的文件路径名称不存在。

④AX＝4，表示打开文件太多，文件代号已用完。

⑤AX＝5，存取无效。可能表示尝试打开一个只读文件或为子目录名称或磁盘标名。

⑥AX＝0CH，不合理的存取模式。

10.5.3 3EH：关闭文件

AH＝3EH，此服务程序可关闭一个文件，并将欲关闭文件代号放在 BX 即可。DOS 会冲洗（flush）与文件代号相关联的文件缓冲区并将数据回写入磁盘及更新目录的内容，并归还此文件代号给 DOS 以供其它新打开的文件使用。

使用实例：

```
mov AH，3EH
mov BX，filehandle
int 21H
jc error
…
filehandle word?
```

执行完后 CF＝0 表成功。若 CF＝1 表失败；回传错误代码在 AX。唯一可能的错误为 AX＝6，无效的文件代号。可能是此文件代号并没有指向一个已打开的文件。有没有想过若关闭的是系统（DOS）所自动打开的 5 个文件代号，会是什么结果。

如下关闭文件代号为 0 的键盘：

```
mov ah，3Eh
mov bx，0   ；关闭键盘
int 21h
```

这将锁死键盘，无法再做输入了。这可是要冷启动（RESET）才能再重新使用。所以记住没有很好理由，不要随意关闭这五个（0，1，2，3，4）文件代号。

10.5.4 3FH：读取数据

AH＝3FH，此服务程序可读取和 BX 中的文件代号相关的文件或装置（就当作是文件使用一样）。

输入参数	意　义
AH＝3FH	
BX	文件代号
CX	所指定欲读出的 byte 数
DX	DS：DX 指向被读取数据所要放置的缓冲区地址

回传参数	意　义
CF＝0	表示读取成功。AX＝真正读到的 byte 数，AX＝0 表读到文件尾
CF＝1	表示读取失败。AX＝错误码

当 OPEN 一个文件时，文件的读写指针是指向文件最开头。当读写完后，指针会跟着移到下一个 BYTE 位置。若回传的 AX 包含 0 或小于指定读取的 CX 数，则表示已到达文件结尾。

使用实例：

```
mov AH，3FH
mov BX，filehandle
```

```
mov CX，512
mov DX，input _buffer
int 21H
jc error   ；跳到错误处理标记 error
cmp AX，CX
jb eof _file   ；跳到 eof _file 标记
```

若指定的文件代号是零(即标准输入 KEYBOARD)，执行时就和 0AH 服务程序一样，可以用 BACKSPACE 和方向键去编辑，按 ENTER 键去结束。CX 为最大的输入字符数。不过 DOS 只允许最多输入 127 个字符。若输入的字符超过 CX 所指定的数目，则只会存储 CX 所指定的数目。结束输入后，DOS 会在此字符串结尾加入 CF/LF 两个字符 (0DH，0AH)。但是并不计算在 CX 中。

使用实例：

```
mov AH，3FH
mov BX，0
mov CX，127
mov DX，offset input _buffer
int 21H
```

可能发生的错误：

①AX＝5，拒绝读取。可能是文件打开是在写入（只写）模式。

②AX＝6，无效的文件代号。可能是文件代号并不指向一个打开的文件。

10.5.5　40H：写入数据

AH＝40H，此服务程序会将 BX 中所指定的文件代号，将数据写入文件或装置。若成功，则 AX＝所写入的字节数。

输入参数	意　　义
AH＝40H	
BX	文件代号
CX	预定写入的字节数目
DX	DS：DX 指向输入数据的开头地址
回传参数	**意　　义**
CF＝0	表成功。AX＝所写入的字节数
CF＝1	表失败。AX＝错误码

CF＝0，AX＝CX，表示写入操作完全成功。

CF＝0，AX＜CX，表示磁盘空间不够，并没有完全写入磁盘。

使用实例：

```
mov AH，40H
mov BX，filehandle
mov CX，256
```

```
mov DX, offset output _ buffer
int 21h
jc error
cmp AX, 256
jNE close _ file   ; 磁盘已满
...
output _ buffer byte 256 dup (?)
filehandle word?
```

有一个有趣的现象是若 CX＝0 而调用 40H 的功能，则没有任何数据写入文件，一旦关闭文件后，文件长度会设为当时读写指针所指的文件位置大小。假设打开文件后，调用 40H 且 CX＝0，关闭文件后，将会发现文件为零。若原文件内有数据则将丢失，全部不见了。所以需特别小心不要不小心更改了文件长度。除非要故意设置文件长度，否则我想一般人是不会随意将 CX 设为零。

可能产生的错误：

①AX＝5，拒绝存取。可能是打开的文件是以只读方式打开或是文件本身属性是只读属性。

②AX＝6，不合法的文件代号。可能是指定的文件代号不是一个已打开的文件。

10.5.6 42H：移动文件读写指针

AH＝42H，此服务程序可移动文件的逻辑读写指针至指定的文件位置上。移动的距离是以字节 (byte) 为单位，由 CX：DX 设置，CX＝读写距离的高 word，DX＝读写距离的低 word。移动的方式由 AL 设置。

若执行成功，CF＝0，DX：AX 将包含文件读写指针的当前位置 (DX 为高 word，AX 为低 word)，代表由文件开头之处至目前指针处的字节数。若失败 CF＝1，AX＝错误码。

输入参数	意　义	
AH	42H	
AL	0	表示从文件起始位置开始移动
	1	表示从文件现在位置开始移动
	2	表示从文件末尾位置开始移动
BX	文件代号	
CX，DX	CX：DX　为指定移动的偏移	
	CX＝位移上半部，DX＝位移下半部	

回传参数	意　义
CF＝0	表成功，DX：AX 传回由文件开头算起的文件指针绝对位置
CF＝1	表失败，AX＝错误码

这种操作与 C 语言 lseek 函数很相似。文件打开时读写器是指向文件开头位置 (0)，我们使用此功能在文件任何位置上作随机读写的操作。

若我们将 AL 设为 2，而 CX：DX＝0000：0000H，可以移动读写指针至文件尾，此时读

写指针（DX：AX）刚好传回文件长度。我们可以由此附加数据至文件尾端（Append to a File）。

使用实例：

```
mov AH，42H
mov AL，2        ；移动至文件尾
mov BX，handle
mov CX，0
mov DX，0
int 21H
mov AH，40H      ；将 buffer 位置起，共 100H 的数据附加至文件尾
mov CX，100H
mov DX，offset buffer
int 21H
…          ；并没有检查写入是否正确
buffer byte 100H dup (?)
handle word?
```

※ 若想知道文件指针当前指向何处，可设 AL＝01H，CX：DX＝0000：0000H，回传的 DX：AX 便是当前文件指针的位置。

使用实例：

```
mov AH，42H
mov BX，handle
mov CX，0
mov DX，0
mov AL，01H
int 21H
jc error
mov word ptr file _position+2，DX
mov word ptr file _position，AX
…
hadle              word?
file _position dword?
```

※ 要注意若 AL＝1 或 2，则移动位移的值可能有正或负数。可以利用此特性，将刚才已写入的数据，再读出，以验证写入是否正确。

使用实例：

```
mov AH，40H
mov BX，handle
mov CX，10
mov DX，offset buffer
int 21H
```

```
jc error
mov AH，42H
mov AL，1
mov CX，0
mov DX，－10
int 21H
mov AH，3FH
mov CX，0
mov DX，offset buffer2
int 21H      ；比较 buffer1 与 buffer2 是否相同
```

可能发生的错误：

①AX＝01H，无效函数号码。可能是 AL 指定 0，1，2 以外的值。

②AX＝06H，无效的文件代号。可能是文件代号所指的文件并未打开。

附录 A　安　　装

1. 系统需求：MASM 6.1 需要下列的系统设计：

(1) 一台 IBM PC 或兼容机。MS-DOS 3.3 或以上。

(2) 80386 或以上 CPU。

(3) 4MB RAM。

(4) 至少需要 5.9MB 硬盘空间（按照安装时你选择 INSTALL 的选项而定）。

(5) 一个 5.25in 软盘驱动器（1.2MB）或一个 3.5in 软盘驱动器（1.44MB）。

2. 安装（INSTALL）和使用 MASM

在“DISK1”中有一个 SETUP.EXE 文件，你可以在 DOS 命令行下或 WINDOWS 中执行“DISK1”中的 SETUP.EXE 来安装你的 MASM 6.11。

(1) 在 DOS 的命令行下安装

将“DISK1”磁盘插入 A 或 B 驱动器中，在 DOS 命令行下键入

A：\SETUP 或 B：\SETUP 即可。

(2) 在 WINDOWS 中执行安装

同样将“DISK1”的磁盘插入驱动器中，打开文件管理器，切换到 A 或 B 驱动器中找到 SETUP.EXE 执行即可。

原始 MASM 6.11 的文件都是经过压缩的，执行 SETUP.EXE 时会自动替您解压缩并将文件 COPY 到你的硬盘。安装的过程非常简单，只需要一直按 ENTER 及陆续放入 DISK2，3，至磁盘中即可安装成功。在安装中有一些选项是可以选择性的，以下是它的缺省值可供参考：

- Load utilities for use with the Windows operating system (default=yes)
- Load the Programmer's WorkBench (default=yes)
- Configure PWB with BRIEF-compatible environment commands (default=no)
- Load MASM.EXE for MASM version5.10 compatibility commands (default=yes)
- Copy Help files, README.TXT, and other documentation files (default=yes)
- Copy the sample programs (default=no)
- Copy a mouse driver (default=yes)
- Select the drive where you want the MASM files to reside (default=highest drive letter)
- Select the directories for the MASM files you choose to install.

These include:

- Executable files (default=C:\MASM 611\BIN)
- Library files (default=C:\MASM 611\LIB)
- Include files (default=C:\MASM 611\INCLUDE)
- Initialization files (default=C:\MASM 611\INIT)
- Help files (default=C:\MASM 611\HELP)

• Sample files (default=C：\MASM 611\SAMPLE)

• Temporary files (default=C：\MASM 611\TMP)

3. 系统文件

SETUP 并不会修改你的系统文件 (CONFIG.SYS, AUTOEXEC.BAT)。为了你的系统能符合 MASM 的设置，SETUP 会将 NEW-VAR.BAT，NEW-CONF.SYS，NEW-SYS.INI 三个文件拷贝到您的\MASM611\BIN 目录下。

(1) NEW-CONF.SYS

NEW-CONF.SYS 是一个 CONFIG.SYS 文件的样本，你可以将以下的内容加到您的系统中，以便能正确的使用 MASM。以下是 NEW-CONF.SYS 文件的内容，其中的数值是最基本的要求，请务必设置大于或等于它的值，如 files=20，您可以更改为 30，40，…，但建议至少要 20。

```
files=20
buffers=10
device=C：\MASM611\BIN\himem. sys
device=C：\MASM611\BIN\emm388.exe 8192 RAM
REM Use the following line to run CodeView without Windows or 386MAX loaded.
REM device=C：\MASM611\BIN\emm386.exe 2048 RAM
```

REM 是注解之意，意即最后一列是当不需要在 WINDOWS 下执行 CodeView 时，可以设 8192 为 2048 就足够了。

(2) NEW-VARS.BAT

NEW-VARS.BAT 是作为设置 MASM 环境变量之用。您可选择性地加入您的 AUTOEXEC.BAT 文件中。以下是此文件的内容：

```
SET PATH=C：\MASM611\BIN；% PATH %
SET LIB=C：\MASM611\LIB
SET INCLUDE=C：\MASM611\INCLUDE
SET INIT=C：\MASM611\INIT
SET HELPFILES=C：\MASM611\HELP\*.HLP
SET TMP=C：\MASM611\TMP
```

% PATH %的意思是，若您原 AUTOEXEC.BAT 文件已设有 PATH，若要再增加 PATH 则需在新增加的列尾加上（%PATH%），才会将新设置的 PATH（C：\MASM611\BIN）加到原来的 PATH 中。如果您能直接加在原来的 PATH 的那一列之后那就不必加%PATH%了。

①PATH=C：\ET3；C：\DOS；

PATH=C：\MASM611\BIN，%PATH%

```
加上%path%可将 c：\masm611\bin 的路径附加至上一列的 path 中
```

②PATH=C：\ET3；C：\DOS；C：\MASM611\BIN；

以上两种皆可，建议使用第一种，因为若一列 PATH 写不完，可利用①的方法再加一行或两行皆可。

（3）NEW-SYS. INI

在 WINDOWS 的目录下您可找到一个 SYSTEM. INI 文件，您可将 NEW-SYS. INI 的内容加在一个以 [386. enh] 为开头的段落中。为了 MASM6. 11 能在 WINDOWS 下顺利执行，增加下列的内容是必须的。以下是此文件的内容：

```
; Add these lines to the [386enh] selection of the SYSTEM. INI file in
; your Windows directory.  The changes will not take effect until
; you exit and restart Windows.
;
; If your SYSTEM. INI file already contains DEVICE=statements with
; the same filenames, replace the old DEVICE=statements with the ones
; shown here.  Windows 3.  x will not run if the SYSTEM. INI file contains
; more than one DEVICE=statements for a single driver.
;
; If your SYSTEM. INI file contains a DEVICE=statement for CVI. 386,
; a driver which is no longer necessary, Windoes 3.  x will not run.
; Remove the DEVICE=CV1. 386 statement to solve this problem.
device=C: \MASM611\BIN\dosxnt.  386
device=C: \MASM611\BIN\cvw1. 386
device=C: \MASM611\BIN\vmb.  386
```

上几例 ‘;’ 之后都是注解，只要将最后的三列的内容加入即可。如果原 system. ini 中有 DEVICE=CV1. 386 这一列，请将之删除，以免 Windows 不能执行。以上这些设置都需要重新开机之后，才能生效。

4. 在 Windwos 中加入 MASM 6. 1 的程序群组（PROGRAMGROUP）

首先，你要确定在 MASM611\BIN 目录下有 MASM. GRP 文件存在。

加入 MASM6. 1 程序群组的步骤：

（1）先进入 Windows，打开文件管理器（Program Manager）

（2）从 File 选单中，选择 New（若中文版为打开文件）

（3）选择程序（中文版）或 Program Group（英文版）

（4）选择 OK 按钮（英文版）或确定（中文版）

（5）键入 C：\MASM611\BIN\MASM. GRP 并按 ENTER，若为中文版请在文件中输入 C：\MASM611\BIN\MASM. GRP。

执行完后您将看到多了一个 MASM6. 1 的程序，其中包含了 Programer's WorkBench、MASM6. 1 Reference (Help 说明)、CodeView，MS-DOS CodeView，Server 共 5 个程序项目。要执行前，需确定 NEW-SYS. INI 中的内容（共三列）已加入 SYSTEM. INI 中，且需马上退出 Windows 及将新的设置值存入文件（在文件管理器选项 [0] 中可找到），再重新启动 Windows 才可执行。

5. 设置 MASM6. 1 开发环境

MASM 的 assembler（编译程序）需要 Extended memory 和 XMS 内存管理程序（如 Himem. sys）。在执行 MASM 时，系统将需要几乎 3.5MB 的扩展内存（含传统内存约共 4MB）。

为了在执行 MASM 和其它程序能有较多的可用内存，若要内存使用较有效率及加速您的程序执行速度，您必须去修改您的 CONFIG.SYS 和 AUTOEXEC.BAT。

当您完成了这两个文件的修改之后，记得重新开机，以使这些修改有效。在 3.x 及 4.x 版的 DOS，在 CONFIG.SYS 中使用 REM 将会产生警告信息，也不能使用 DEVICEHIGH 这个命令。在 AUTOEXEC.BAT 中也不能使用 LOADHIGH 命令。切记！切记！建议改用 DOS 5.0 或以上版本即可解决。

6. 修改 AUTOEXEC.BAT

AUTOEXEC.BAT 中包含有许多环境变量的设置及意义。为了能让 MASM 能在最理想的状态下工作，这些环境变量是需要设置的，大部分的设置在 NEW-VARS.BAT 都有包含。

（1）需要的环境变量

1）INIT　指定 TOOLS.INI 和 CURRENT.STS 的目录。如果 INIT 环境变量有指定的话，PWB 会依照 INIT 指定的目录去寻找 TOOLS.INI（initialization）和 CURRENT.STS (status)；如果没有指定 INIT 变量，则 PWB 和 CodeView 在每一个目录自动建立 CURRENT.STS，而使得下次执行时的状态可能和上一次不同。

2）PATH　指定搜寻 EXE 文件的路径。

3）TMP　指定暂时文件存储的目录。像 NMAKE，LINK 和 PWB 这些公用程序 (utilities) 都会使用到 TMP 环境变量。

4）HELPFILES　当执行 PWB，CodeView 和 QuickHelp 需要 Help 时，所指定 Help 文件放置的路径。

（2）选择性的环境变量

1）LIB　指定 library 文件被放置的目录。

2）INCLUDE　指定 INCLUDE 文件被放置的目录。

3）MASM　放置兼容 MASM5.10 的目录。

4）ML　指定 ML 放置的目录。

5）LINK　指定 Linker 放置的目录。

（3）使用 TMP 和 TEMP 环境变量

Microsoft 程序语言或公用程序都会设置 TMP 目录去当作寄存文件存储区段。例如 SET =C：\MASM611\TMP 或 SET TMP=C：\TMP。

例如 PWB 执行时就至少需要 1MB 的磁盘空间在 TMP 目录下，因为需放置 PWB 的虚拟内存文件（Virtual memory files）。像 Microsoft 应用程序（如 Word）所使用的环境变量是 TEMP，而 Microsoft 程序语言所使用的 TMP 变量，都是用来当作暂时文件存储空间。不过一般应用程序（Applications，AP）所需使用的磁盘空间较 TEMP 小，所以您可将 TEMP 设置在如 RAM 磁盘中，而 TMP 一般都是设置在硬盘中。

使用环境变量有几点要注意：

·在 SETUP 时如果之前在 AUTOEXEC.BAT 中没有设置 TMP 变量，SETUP 会要求你输入一个路径目录去作为暂时文件存储之处（缺省值为\MASM611\TMP）。你当然可以自

已指定一个路径，最好是这个路径目录已存在。而在安装完 MASM6.11 后，会在 NEW-VARS.BAT 文件中有一列 SET TMP=\MASM611\TMP，当然得自己动手将它加入 AUTOEXEC.BAT。

·注意 TMP 环境变量如果是指到一个网路磁盘，必须确定这条路径没有防写保护，否则会造成无法执行 MASM。

·在说明一次 TMP 环境变量所指定的变量要存在，若不存在则 SETUP 执行时会将根目录作为暂时存储文件的地方。这会产生一个问题就是 MS-DOS 会限制在根目录文件建立数目。

(4) 修改 CONFIG.SYS

同样地 SETUP 并不会修改您的 CONFIG.SYS，不过会将建议更改的地方写入 NEW-CONF.SYS(MS-DOS)，NEW-SYS.INI(SYSTEM.INI from Windows)中。在 C:\MASM611\BIN 目录下可找到。SETUP 会加入 BUFFERS，FILES，DEVICE 命令到 NEW-CONF.SYS 中，依照您的系统设置，为了 MASM 能在您的系统中顺利执行许多更改是必须的。

1) BUFFERS

BUFFERS 这个命令是使用在 CONFIG.SYS 中，系统启动时缺省的 BUFFER 数为 15。用户可用 BUFFER=N 来指定 BUFFER 数。若磁盘的目录组织较复杂可以修改 CONFIG.SYS 指定较多的 BUFFER 数，使磁盘的读写效率较高。每个 BUFFER 会占用 532 bytes，若设置 BUFFER 过多，相对地用户的应用程序减少了可用的内存空间，反而造成数据区空间不足频频读写的情况发生，执行速度也会减慢下来。以下是建议值。

硬盘大小	Buffer 数
40MB 以下	20
40～79MB	30
80～119MB	40
120MB 以上	50

⊙如果你有使用 SMARTDRV.EXE，建议将 BUFFER 数设为 10。因为 SMARTDRV.EXE 比 BUFFERS 的效果更佳。

2) FILES

FILES 是设置同一时间内能存取的 FILE 数。每一个 FILE 数会占用 48 bytes 的内存。在 MS-DOS 下执行 MASM 建议至少 FILE 数=20。若在 WINDOWS 下建议 FILES=30。若不指定，DOS 缺省值为 8。

3) DEVICE、DEVICEHIGH

DEVICE 命令会载入一个装置驱动程序。DEVICEHIGH 会将装置驱动程序载入上层内存。若使用 DEVICEHIGH 命令则需加入 DOS=UMB。DEVICE 则需 HIMEM.SYS 及 EMM386.EXE。

(5) 修改 SYSTEM.INI

同样 SETUP 时并不会修改 SYSTEM.INI(执行 Windows 时会用到)，在 C:\MASM611\BIN\NEW-SYS.INI

有三条语句是必须加在 SYSTEM.INI 中的 [386enh] Section 中。内容如下：

DEVICE=C：\MASM611\BIN\DOSXNT.386

DEVICE=C：\MASM611\BIN\CVW1.386

DEVICE=C：\MASM611\BIN\VMB.386

如果你想在 WINDOWS 的 MASM6.1 下执行 WX.EXE，VMB.386 是一定要加入的。

（6）修改 TOOLS.INI

当执行 PWB 及 CodeView 时需使用到 TOOLS.INI。执行 PWB 时一般设置都记录在 TOOLS.INI。而 CodeView 执行时也会使用到 TOOLS.INI 的［CV］及［CVW］Section 内的设置。而且 TOOLS.INI 必须被放置在 INIT 环境变量所指定的目录中。若您没有 TOOLS.INI 时，没关系 SETUP 时会载入 TOOLS.PRE 在 C：\MASM611\INIT 路径目录中，您只需 copy TOOL.PRE 到 TOOLS.INI 即可。

（7）使用 DOSXNT.EXE

在 C：\MASM611\BIN（缺省值）目录下有一 DOSXNT.EXE，若您在 MS-DOS 下要使用 MASM 编译器；则必须要将 DOSXNT.EXE 放在当前的工作目录或 path 所指定的路径下。

附录 B　LST、REF、MAP 文件

1. 产生和读列表文件

一个列表文件精确地显示编译程序如何将你的原始文件翻译成机器码。利用 MASM 有三种情况可以建立列表文件：

□在 PWB 中选择适当的选项。

□在原始文件中使用相关联的 directives。

□在 MASM 命令行中指定/F1。(是小写的 L 不是一)

在列表中包含在原始文件中的语句和每一个语句产生的二进制码。在最后还会列出原始文件中出现的 names，和所有的标记、变量和符号。

编译程序会为 macros，structures，unions，records，segments，groups，和其它 symbols 建立表格在列表文件的结尾。不过原始文件中须有使用到才会出现在列表文件中。例如：如果你的程序没有宏（macros），符号表就不会有宏这一节。

2. 产生列表文件

要在 PWB 产生列表文件首先有几个步骤需做完：

(1) 从 Options menu 中的 Language Options 选择 MASM Options。

(2) 在 MASM 对话框中选择 Set Debug 或 Set Release Options。

Set Debug 或 Release Options 的对话框中列出的选项摘要列于表 B-1。此表也列出可使用的原始文件中对等的 directives 和命令行参数。

表 B-1　产生或修改列表文件的参数

To generate this command		In source	From
information	In PWB[1], select:	code, Enter:	lines, Enter:
Default listing-includes all assembled lines	Generate Listing File	.LIST (default)	/FI
Turn off all source listings (overrides all listing directives)	Generate Listing File (turn off)	.NOLIST (synonym=.SFCOND)	—
List all source lines including false conditionals and generated code	Include All Source Lines	.LISTALL	/FI/Sa
Show instruction timings	Listing Instruction Timings	-	/FI/Sc
Show assembler-generated code	List-Generated Instructions		/FI/Sg
Include false conditionals	List False Conditionals	.LISTIF (synonym=.LFCOND)	/FI/Sx

（续）

To generate this command		In source	From
Suppress listing of any subsequent conditional blocks whose condition is false	List False Conditionals (turn off)	.NOLISTIF (synonym=.SFCOND)	—
Toggle between .LISTIF and .NOLISTIF	—	.TFCOND	—
Suppress symbol table generation	Generate Symbol Table (turn off the default)	—	/FI/Sn
List all processed macro statements	—	.LISTMACROALL (synonym=.LALL)	—
List only instruction, data, and segment directives in macros	—	.LISTMACRO (default) (synonym=.XALL)	—
Turn off all listing during macro expansion	—	.NOLISTMACRO (synonym=.SALL)	—
Specify title for each page (use only once per file)	—	TITLE name	/St name
Specify subtitle for page	—	SUBTITLE name	/Ss name
Designate page length and line width, increment section number, or generate page breaks	—	PAGE [length, width] [+]	/Sp length /SI width
Generate first pass listing	—	—	/Ep

[1] Select MASM Options from the Options menu, then choose Set Dialog Options from the MASM Options dialog box.

3. 在前的命令行参数与列表伪指令（directives）

我们可以在DOS的命令行下参数和在原始程序中指定相对应的伪指令有相同的效果。编译程序会解释这些命令。

□/Sa 参数会推翻在原始文件中的伪指令（任何会抑制列表的伪指令）。

□除了/Sa 外，在原始文件中的伪指令会推翻所有在命令行下的参数

□.NOLIST 会推翻其它的列表伪指令，如 .NOLISTIF，.LISTMACROALL

□/Sx，/Ss，/Sp，和/SI 参数会对它们各自的特征设置启始值。

4. 读列表文件

在 .DATA 伪指令之后，在列表文件左边区段显示在数据段中的偏移（offset）和起始值（bytes）。

指令是开始在 .CODE 伪指令之后。在列表文件中编译程序在左边产生三个区段（offsets、instruction timings 和 binary code）。缺省值只有 offsets 与 binary code。

在右边区段列出真正在原始文件中出现的语句（statements）或由宏展开的语句。在中间

的区段有许多符号和简体字提供有关程序码的信息。

5. 产生程序码

编译程序会列出由原始文件中的语句产生的程序码。用/Sc 参数在命令中将产生 instruction timings，如下：

offset [timing] [code]

offset 是从目前代码段开始的偏移。timing 表示此处理器执行指令所需要的周期数（cycles）。timing 值反映了 CPU 的形式；例如，指定 .386 伪指令产生对于 80386 处理器的 instruction timings。如果语句产生 code 和 data，code 显示十六进制数值（如果在编译时间 assembly time 就知道这个数值）。如果此数值被计算在 run time，编译程序会表示需要再计算此数值。

当在缺省的 .8086 directive 之下编译时，timing 表示一个有效地址（如果此指令存取内存）。80186/486 处理器不使用有效地址值。

当在缺省的 .8086 directive 之下编译时，timing 表示一个有效地址（如果此指令存取内存）。80186/486 处理器不使用有效地址值。

6. 错误信息（Error Messages）

如果在编译过程中有任何错误发生，而在错误发生的语句下面直接会有错误信息和错误号码出现。实例如下：

mov ax, [dx] [di]

example. asm (17): error A2031: must be index or base register

7. 符号与简体字（Symbols and Abbreviations）

编译程序使用符号与简体字去表示必需在 link 时才能解决的地址或一个必须用特别的方法产生的值。表 B-2 列出所有的符号与简体字：

表 B-2 Symbols and Abbreviations in Listings

Character	Meaning
C	Line from include file
=	EQU or equal-sign (=) directive
nn [xx]	DUP expression: nn copies of the value xx
---	Segment /group address (linker must resolve)
R	Relocatable address (linker must resolve)
*	Assembler-generated code
E	External address (linker must resolve)
n	Macro-expansion nesting level (+if more than 9)
|	Operator size override
&	Address size override
nn:	Segment override in statement
nn/	Rep or lock prefix instruction

表 B-3 解释在 timing 区段符号的意义

表 B-3 Symbols in Timing Column

symbol	Meaning
m	Add cycles depending on executed instruction
n	Add cycles depending on number of iterations or size of data
p	Different timing value in protected mode
+	Add cycles depending on 操作数 s or combination of the preceding
,	Separates two values for "jump taken."

8. 列表文件中表格的解释

在列表文件结尾会列出许多表格如 macros，structures，unions，records，segments，groups，和 symbols（当然是曾经出现在程序中的）。

（1）Macros Table

列出在主要程序或包含文件中所有宏（macros）。

（2）Structures and Unions Table

提供 structure 或 union 每个区段的 offset（bytes）。

（3）Record Table

"Width" 指明整个 record 所占的 bit 数。"Shift" 提供从 record 最低位到此区段最低位的偏移。每个区段的 Width 指明每个区段的 bit 数。"Mask" 指明区段的最大值（十六进制）。

"Initial" 指明每个区段的初始值。

（4）Type Table

"Size" 指是有 TYPEDEF type 的大小 (bytes)。"Attr" 指明对于 TYPEDEF 定义的基本 type。

（5）Segment and Group Table

"Size" 指明段是 16 位或 32 位。"Length" 指明段所占的大小（bytes）。"Align" 指明段的边界（alignment)；如 WORD，PARA 等等。"Combine" 指明段的结合型式（PUBLIC，STACK 等等）。"Class" 指明段的类别（CODE，DATA，STACK，或 CONST）。

（6）Procedures，Parameters，and Locals

指明定义在每个 procedure 中所有的 parameters 和 locals 从 BP 算起的 offset 与 types，每个 procedure 的内存位置与大小。

（7）Symbol Table

所有的 symbols（除了 names 为 macros，structures，nuions，records，和 segments）都会列在列表文件结尾的符号表中。"Name" 是以字母的顺序排列。如果是一个多重元素的变量的长度（如数组或字符串）是指单一元素的长度，不是指所有变量的长度。

如果 symbol 是表示一绝对值（如用 EQU 定义或 "=" 伪指令)，则 "Value" 区段会显示它的值。如果 symbol 是表示变量或标记，则 "Value" 区段显示从它被定义的段开头地址算起的十六进制偏移。

"Attr" 区段显示 symbol 的属性，包含 symbol 被定义的段名字，symbol 的可视性 (scope)，code 长度。symbol 的可视性只有在使用 EXTERN 和 PUBLIC directive 定义时才会

有。Scope 可以是 external，global，或 communal。如果区段是空的，表示没有属性。

9. 细说列表文件

下面以第 3 章 3.4 节的实例程序的列表文件做例子来解释列表文件的结构。我们假设实例程序文件名为 example. asm，在 DOS 命令行下执行

ML/FI /Sg /Sc example. asm

就可产生 example. lst。/Sg 是表示要列出编译程序所替我们产生的程序码，如利用简化伪指令（.STARTUP，.EXIT）。/Sc 是表示要列出各指令的 timing。缺省值是不会产生的，也就是/Sa，/Sc 是选择性使用的参数。右边是原始程序，左边是由 assembler 产生的机器语言，数据区段和 timing。所有的数值皆以十六进制表示。Assembler 将每一新区段开始在 0000H（零）的偏移（bytes），所以最大至 64KB。

总共 5 页。图 B-1 是 example. lst 列表文件的第 1 页（Page 1.1）。程序的标题（Title）列在每页的第 2 列。首先开始的是数据段（.DATA），开始的 offset 是“0000”。

左边第 1 个区段是偏移，可以看到 message _ length 在偏移“0016”（22 Decimal），第 2 个区段是字符的 ASCII 值。如 48H（字符“H”）在偏移 0000H，65H（字符“e”）在偏移 0001H。EQU 伪指令是告诉 assembler 此 message _ length 被给定和 message 的长度相等的值（0016H）。

图 B-1：example. lst 列表文件的第 1 页

```
Microsoft (R) Macro Assembler Version 6.11          07/25/95 02:55:44
DISPLAY A MESSAGE                                   page 1-1
                                        PAGE 66,80  ;程序的目的是要在屏幕上印出一列信息
                                        TITLE DISPLAY A MESSAGE
                                        .MODEL SMALL
                                        .386
                                        .DOSSEG
                                        .STACK 1024
0000                                    .DATA
0000 48 65 6C 6C 6F 2C 20 68                message DB 'Hello, how are
             6F 77 20 61 72 65
             20 79 6F 75 20 3F
             0D 0A
0016=0016                                   message _ length EQU $-message
0000                                    .CODE
```

最后一列为 .CODE 段的开头。因为是一个新区段，所以偏移重设为 0000H。因下一列为 PAGE 伪指令，所以下面内容从第 2 页开始。

图 B-2 为第 2 页内容。可以看到 PAGE 伪指令并没有产生任何机器码。因为它只是对编译程序所下的伪指令，并不会产生任何机器指令。第一个指令开始在偏移 0000H，第二个指令开始在偏移 0003H，第三个指令开始在偏移 0005H 等等。最后一个指令开始在偏移 0020H（32Decimal）。因此，此代码段（code segment）的范围是从 0000H 到 0022H。所以代码段共

占 22H（34Decimal）bytes。

在偏移右边的区段是 timing 数，我们看到偏移 001CH 与 0020H 的指令（INT 21H）的 timing 数为 37。在偏移 0003H 与 000CH 有符号"p"的 timing 是指在保护模式下有不同的 timing 值。

在偏移右边的区段是真正被产生的机器指令。每个机器指令的第一个 byte（以两个十六进制数字表示）是 opcode (operation code)。例如在偏移 0020H 的 INT 的 opcode 是 CDH，操作数（操作数）是 21H。

大部分的指令都有操作数。操作数可以是寄存器，例如：

MOV AH，40H ；如 AH 是一目的操作数

操作数可以是变量（内存位置的名称），例如：

LEA DX，message ；message 是一变量

操作数可以是真正值，又称立即操作数（immediate 操作数），例如：

INT 21H ；21H 是一立即数

在偏移 0015H 的机器指令

B9 0016 ；B9 是 opcode，0016 是…真正值（message _ length）

在偏移 0018H 的机器指令

8D 16 0000 R

因为操作数的完整地址（full address）是 DS：message（记住，处理器是假设 DS 是此段寄存器，除非你指定其它）。因为此地址须依照 DS 的值，须等到程序载入 memory 才知道，所以此操作数是 Relocatable。注意"R"是代表"Relocatable"重定位，指的是此地址无法完全决定要直到程序载入 memory 执行时才能决定。

这些地址被称为"relocatable"是因为使用这样的地址可以被载入内存的任何地方（重定位；relocated）。

现在看看在偏移 0000，操作数是以"----"表示。assembler 使用"----"去表示一个区段的地址。注意此地址也是需重定位 (Relocatable)。最右边列出原始程序的地方，前有"*"的地方是代表编译程序产生的 code，不是我们所写的。

图 B-2：example. lst 列表文件的第 2 页

```
Microsoft (R) Macro Assembler Version 6.11          07/25/95 02:55:44
DISPLAY A MESSAGE                                              Page 2-1

                                        PAGE
0000                                 main PROC
                                        .STARTUP
0000                          *  @Startup:
0000  2  B8----R              *           mov   ax, DGROUP
0003  2p 8E D8                *           mov   ds, ax
0005  2  8C D3                *           mov   bx, ss
0007  2  2B D8                *           sub   bx, ax
0009  3  C1 E3 04             *           shl   bx, 004h
```

```
000C   2p 8E D0                    *          mov  ss, ax
000E   2   03 E3                   *          add  sp, bx
0010   2   B4 40                              MOV AH, 40H
0012   2   BB 0001                          MOV BX, 0001H
0015   2   B9 0016                          MOV CX, message_length
0018   2   8D 16 0000 R                      LEA DX, message
001C 37    CD 21                                INT 21H
                                           .EXIT
001E   2   B4 4C                *          mov  ah, 04Ch
0020 37    CD 21                *          int  021h
0022                                       main ENDP
                                   END
```

图 B-3 是包含程序中所定义的段与群组名称的信息。

图 B-3：example. lst 列表文件的第 3 页

```
Microsoft (R) Macro Assembler Version 6.11          07/25/95 02:55:44
DISPLAY A MESSAGE                                        Symbols 3-1


Segments and Groups:
                Name          Size      Length    Align   Combine Class

DGROUP          ..........    GROUP
_DATA           ..........    16 Bit    0016      Word    Public  'DATA'
STACK           ..........    16 Bit    0400      Para    Stack   'STACK'
_TEXT           ..........    16 Bit    0022      Word    Public  'CODE'
```

我们看到 STACK 定义了一堆栈段，长度是 0400H（1024 decimal），且起始边界是一个 RARAGRAPH 边界。

图 B-4 是包含程序中所定义的 Procedures，Parameters 和 locals 的信息。

main 是一个 procedure，而 type 是“P Near”代表“Procedure Near”。Near 表示此 Procedure 是在它自己的段被调用（called）。

@Startup 是一个 label，而“L Near”是“label Near”，表示是一简单地址。如果不是 Near 而是 Far，表示可以从另一段调用它。

图 B-4：example. lst 列表文件的第 4 页

```
Microsoft (R) Macro Assembler Version 6.11          07/25/95 02:55:44
DISPLAY A MESSAGE                                        Symbols 4-1


Procedures, parameters and locals:
                Name          Type      Value     Attr
```

```
main            ..........   P Near    0000    _TEXT   Length=0022 Public
@Startup        ..........   L Near    0000    _TEXT
```

图 B-5 是包含程序中所定义的 Symbols 的信息。

message _ length 是一个值为 0016H 的 Number。message 是一个在数据段（_DATA）位置为 0000H · BYTE 的地址。

最后在列表文件结尾处有警告（Warnings）与错误（Errors）数的统计。意指错误比警告较严重。如果你的程序有 errors 请不要使用此 obj 模块。大部分时间，你应该修改警告和错误后再重新编译（reassemble）一次才是正确之途。

图 B-5：example. lst 列表文件的第 5 页

```
Microsoft (R) Macro Assembler Version 6.11          07/25/95 02:55:44
DISPLAY A MESSAGE                                        Symbols 5-1


Symbols:
                Name            Type     Value    Attr
@CodeSize  ..................   Number   0000h
@DataSize  ..................   Number   0000h
@Interface  .................   Number   0000h
@Model  .....................   Number   0002h
@code  ......................   Text     _TEXT
@data  ......................   Text     DGROUP
@fardata?  ..................   Text     FAR_BSS
@fardata  ...................   Text     FAR_DATA
@stack  .....................   Text     DGROUP
message_length  .............   Number   0016h
message  ....................   Byte     0000     _DATA

        0 Warnings
        0 Errors
```

10. 建立 REF 文件

cross reference 文件会对使用在程序中每一符号名称产生一报告。当你正在除错（debugging）时，这份报告特别有用，尤其在一大程序中有很多符号名称时。

我们在 ML 之后加上/Fr 参数会产生一个 SBR 文件，它是使用在 PWB 的一个信息文件。若想产生一个 REF 文件，需先将 SBR 文件转化为 BSC 文件，执行如下

BSCMAKE EXAMPLE. SBR 或 PWBRMAKE EXAMPLE. SBR

将会产生一个 EXAMPLE. BSC 文件。再执行 CREF 即可产生一个可读的 ASCII 格式的 REF 文件（default）。执行如下

CREF EXAMPLE. BSC，EXAMPLE. REF

将产生一个 EXAMPLE. REF。REF 扩展名是缺省值。所以你可以省略 REF 或者连 EXAMPLE. REF 都可省略不写。当然你也可以将之存在另一文件名中。如 TEST. DOS，只须将 EXAMPLE. REF 改成 TEST. DOC 即可。

还有另一种情况就是只键入 CREF，接着它会问你的 BSC 文件名为何，及要存入哪一文件，缺省的文件扩展名仍为 REF。

11. 读 REF 文件

图 B-6 为 example. ref 文件的内容。

所有在程序中出现的符号名称都以字母顺序列出。右边的数字是此符号出现在程序中的列号，而“#”旁的数字是代表它被定义的地方。

Cross reference 文件用处是当你工作在一个大程序中，要寻找所有符号出现的程序的位置时，提供一个容易的方法。

图 B-6：example. ref

```
Microsoft Cross-Reference Version 6.11      07/25/95 01:25:37

Symbol Cross-Reference              (#definition)          Cref-1

@code
    a. asm ……                      3#

@CodeSize
    a. asm ……                      3#

@data
    a. asm ……                      3#

@DataSize
    a. asm ……                      3#

@fardata
    a. asm ……                      3#

@fardata?
    a. asm ……                      3#

@Interface
    a. asm ……                      3#      20

@Model
    a. asm ……                      3#

@stack
    a. asm ……                      3#
```

```
_DATA
    a. asm ……    3#

_TEXT
    a. asm ……    3#    10

DGROUP
    a. asm ……    3#    13    3

main
    a. asm ……    12#

message
    a. asm ……    8#    17    9

message _length
    a. asm ……    9#    16

STACK
    a. asm ……    6#
```

12. 建立与读 MAP 文件

要建立一个 MAP 文件只需在 ML 之后加上/Fm 参数即可。MAP 文件是由 Linker 所产生的一份报告。大部分的时间你不需要使用它。图 B-7 为一 MAP 文件的实例。

所列的段顺序刚好是程序执行时载入 memory 的顺序，也就是说 CODE 段首先载入，接下来是 DATA 段，最后是 STACK 段。

段的长度是以 BYTE 为单位。CODE 段是 32H（50 Decimal）bytes；DATA 段是 16H（22 Decimal）bytes；STACK 段是 400H（1024 Decimal）bytes。Start 和 Stop 区段是指每一段开始和结束的地址。然而这些地址是相对于此程序执行前被载入 memory 的地址。真正的地址在每次程序执行时可能是不同的。

注意因为 STACK 的起始边界(align)是 PARAGRAPH，所以 STACK 是开始在 00050H，而不是 00048H。(1 paragraph＝16 bytes)。这是因为 Linker 会跳过 00048H，而 DATA 的起始边界是 WORD（1 word＝2 bytes），所以 DATA 段开始在偶数地址 00032H。

图 B-7：example. map

Start	Stop	Length	Name	Class
00000H	00031H	00032H	_TEXT	CODE
00032H	00047H	00016H	_DATA	DATA
00050H	0044FH	00400H	STACK	STACK

Origin	Group
0003：0	DGROUP

```
Address         Publics by Name

0000: 0010      main
0003: 0020      _edata
0003: 0020      _end
0003: 0020      __edata
0003: 0020      __end

Address         Publics by Value

0000: 0010      main
0003: 0020      __edata
0003: 0020      __end
0003: 0020      _end
0003: 0020      _edata

Program entry point at 0000: 0010
```

附录C 完 整 段

使用完整段（Full Segment）定义

如果你需要完整的控制所有段，在你的程序中你需要完整的定义段，而不能使用简化段。接下来将解释段的定义，包含如何安排段和如何定义段的型式。

如果你在 MS-DOS 下面写程序而不使用 .MODEL 和 .STARTUP，你必须自己设置寄存器的初值和使用 END 伪指令去表示起始点。

1. 用 Segment Directive 定义段

一个段定义时，须以 SEGMENT 伪指令开头，ENDS 伪指令结束。

```
name SEGMENT [ALIGN] [readonly] [combine] [use] ['class']
statements
name ENDS
```

name 定义段的名称。连结程序也会结合在不同模块中有相同名称的段，除非结合的型式 PRIVATE。除此之外，段也可以网状化。

在 SEGMENT 伪指令之后选择的 TYPES，是告诉 linker 和 assembler 如何去设置和结合段。所有可选择的 TYPES 解释在下面的各节中，包含有：

Type	说 明
Align	指定段将被载入时在内存中的起始边界
READONLY	告诉 assembler 如果有侦测到有指令去更改到在 READONLY 段的任何项目要返回一个错误
Combine	决定当在建立一个执行文件时，Linker 在不同模块中如何结合段
Use（386/486）	决定一个段的大小。use16 表示在这个段的 offest 是 16 bits 宽。即大小是 64K 的段 use32 表示 32 bits 的 offset
Class	提供这个段的类别名。Linker 会自动将有相同类别名的段连续排列一起写入 EXE 文件

Type 可被指定在任何顺序，但每个段只能指定一种属性，例如你不可以有两个不同的 align types。

你可以关闭一个段稍后再打开（用 SEGMENT 伪指令）。当你再打开一个段时，你只需要给定段名称。一旦你已定义好段之后，不可改变此段的属性。

注意 PAGE align type 和 PUBLIC combine type 是和 PAGE and PUBLIC 伪指令不同的。Assembler 会根据上下文来区分它。

2. Aligning Segments

可选择性的 align type 定义段将来载入时，段开头地址在内存的范围。也就是这个段被定义好要和其它段结合时，告诉 Linker 有多少 bytes 需要去跳过而从 align type 指定的地址开始载入。

align type	开 头 地 址
BYTE	从下一个任意可用的 byte 地址载入
WORD	从下一个任意可用的 word 地址，即由偶数地址载入
DWORD	从下一个任意可用的 double word 地址载入
PARA	从下一个任意可用的 paragraph 地址载入。（每个 paragraph＝16bytes；Default）
PAGE	从下一个任意可用的 page 地址载入。每个（page＝256 bytes）

Linker 使用 align type 去决定每一个段的相对开头地址。当程序要载入 memory 中执行时，操作系统会计算真正的开头地址。

如果程序是在 8086 或 80286 上执行，WORD align type 对于数据段是最好的。这是因为这两种 CPU 有一个 16 位数据总线。变量在一个偶数的地址只需要一个内存提取（memory fetch），而奇数地址需要两个。

如果程序是在 80386 或 486 处理器上执行，DWORD align type 对许多运算是较有效率的。EVEN 伪指令也可以使用在一个段去强迫下一个指令或变量在一个偶数地址，如下例：num3 应被配置在一个奇数地址（如果没有使用 EVEN）。在 num3 要被配置在一个偶数地址之前，MASM 会插入一个 byte（90H）（一个 NOP，或 no operation 指令；会被 CPU 忽略。）

```
exampleseg    segment    word
      num1      word     1000H
      num2      byte       ?
       even
      num3      word     3000H
exampleseg ends
```

3. 使 Segments 有只读（READ-ONLY）属性

可选择的 READONLY 属性对于保护模式要建立只读段是有帮助的，或当程序码要被放置在 Read-Only Memory（ROM）。

READONLY 属性使 assembler 去检查是否有指令去修改段，如果有任何发现将会产生一个 error。如果你尝试去写入 read-only 段，assembler 会产生一个 error。

4. Combining Segments

Combine type 指定在不同模块间出现相同段名称时，Linker 要如何结合这些段。combine type 是控制 Linker 的行为，而不是 assembler。

combine type 包含：

Combine Type	Linker Action
PRIVATE	表示私有段，就算有许多相同的段名称，也不会将段结合在一起
PUBLIC	共用段。表示会连接所有相同名称的段而形成单一连续的段
STACK	会连接所有相同名称的段而形成单一连续的段，并且使操作系统去设置（SS：00）到此段的底部（bottom）和（SS：SP）到顶端（top）
COMMON	重叠段。所有同名的段会使用同一个起始地址。各段的使用空间相互重叠。所有变量的 offset 由相同的起始地址计算，且变量可相互重叠。
MEMORY	与 PUBLIC 相同。为了与 Intel 的 link 格式而设。

（续）

Combine Type	Linker Action
AT address	让你建立一个段的绝对地址，此地址必须是????0H，即是一个 PARA 起始地址。所有变量或数据不可具有初始值，但可定义一个 strncture 或变量去存取一个 for memory 的地址，例如一个屏幕缓冲区（screen bnffer）或低内存（low memory） 例如 keyboard buffer status 是个很好的例子： BIOS segment at 40H org 17H keyboard-status db? BIOS ends

注：不可以使用在保护模式的程序中。

不可放置具有初值的数据在 STACK 或 COMMON 段，因为这些结合的型式（combine type），Linker 会重叠段开头每一模块的起始数据，也就是说最后一个段内的数据会覆盖掉其它模块内的数据。

正常情况下，在一个程序中你应该至少提供一个 stack segment（有 STACK combine type）。如果没有 stack segment 被说明，Linker 会显示一个警告信息。如果你有特别的理由没有说明一个堆栈段，你可忽略这个信息。例如你不应该有一个分离的堆栈段在一个 MS-DOS tiny 模式（COM）程序中，或在 DLL 中使用 caller's stack 时你不需要有一个分离的 STACK。

5. 设置段 WORD SIZES（80386/486 only）

USE TYPE 可用在 80386/486 处理器上，指定一个段的大小。大小的属性可能是 USE16，USE32 或 FLAT。如果你在 .MODEL 伪指令之前指定 .386 或 .486，USE32 是 default。意思是这个段内所有项目的地址是一个 32 位。如果 .MODEL 使用在 .386 或 .486 之前，USE16 是 default。为了确定 USE32 是 default，你应该放置 .386 或 .486 在 .MODEL 之前。你可以再指定一次 USE32 去覆盖缺省值是 USE16 的属性。反过来也可用 USE16 去覆盖缺省值为 USE32 的属性。

MS-DOS 程序不能指定 USE32。16 位和 32 位混用在同一程序中是可能的，但通常是运用在系统程序中。

6. 使用 CLASS TYPE 设置段顺序

可选择性使用的 CLASS TYPE 帮助控制段顺序。如果两个相同名称段的 CLASS（类别）不同，不能结合成一个段。CLASS 是指定 linker 把有相同 CLASS 的段依在 OBJ 文件内的排列顺序连续写入 EXE 文件。CLASS TYPE 是一个简单字符串（不分大小写），用（''）括起来的字符串。有相同 CLASS TYPE 的段会被一起载入，虽然在原始程序中有不同的顺序。

7. 控制段顺序

正常情况下 assembler 安排段位置是依照段出现在 source code 的顺序。而 linker 处理 OBJ 文件的顺序是依照各 OBJ 文件出现在命令行的顺序。

你通常会忽略段的顺序，但有时你可能要某些段出现在程序的开头或结尾，或假设许多

段是依序出现在内存中。段的顺序是重要的。对于 TINY 模式（.COM）程序，CODE 段必须首先出现在执行文件，因为必须在开头地址 100H 之处开始执行。

8. 段顺序 directives

你可以用三个 directives 去控制出现在执行程序的段。.SEQ（default）会依照你说明段的方式去安排段。

.ALPHA directive 会依照字母的顺序安排段。如果你用旧的汇编语言写成的程序执行有麻烦，试试 .ALPHA。它指供对早期 IBM assembler 的兼容性。

.DOSSEG directive 指定 MS-DOS 段按照 Microsoft 语言的标准段顺序放置。不要在子程序中使用 .DOSSEG。

.DOSSEG directive 安排段如下：

（1）Code 段

（2）Data 段，在这个顺序：

a）不具 BSS 或 STACK 类别的段；

b）有 BSS 类别的段；

c）有 STACK 类别的段。

※ BSS 表示不具初值的数据段。

当你说明两个或较多的段（具有相同 CLASS），linker 会自动排列使它们连续，利用这个规则可重新安排段的顺序。

9. Linker 控制

大部分安排段顺序技巧（CLASS，.ALPHA 和 .SEQ）控制 assembler 安排段出现在 OBJ 文件的顺序。通常你较有兴趣的是出现在执行文件中的段顺序。linker 会控制此顺序。

linker 处理 OBJ 文件的顺序是按照它们出现在命令行的顺序。在每一个模块中，是按照它们出现在 OBJ 文件中的顺序。假设在第一个模块说明了 DSEG 和 STACK 两个段，第二个模块说明了 CSEG 段，而 CSEG 是最后输出的。如果你要放 CSEG 在前面，有两种方法去做。

（1）最简单的方法是使用 .DOSSEG，它会告诉 linker 按照 Microsoft 的段顺序去安排段，它可盖掉在命令行中 OBJ 文件的顺序而按照 Microsoft 的格式将所有有 'CODE' 类别的段放在前面。

（2）另一个方法是你可先说明你要安排的段。（例如使用一个包含文件：include file，或在第一个模块中先说明。）也就是你可先说明一个空段（dummy segment），也就是段内没有内容。Linker 会观察这些段的顺序，然后结合这个空的段和在其它模块中有相同名称的段在一起。例如：你可以将下面内容放在第一个模块的开头或是一个包含文件中：

```
_TEXT   SEGMENT   WORD   PUBLIC    'CODE'
_TEXT   ENDS
_DATA   SEGMENT   WORD   PUBLIC    'DATA'
_DATA   ENDS
CONST   SEGMENT   WORD   PUBLIC    'CONST'
CONST   ENDS
STACK   SEGMENT   PAPA   STACK     'STACK'
STACK   ENDS
```

稍后在你的程序中将会按照_TEXT，_DATA，…的顺序排列。

10. ASSUME 伪指令

许多汇编语言指令均是假设一个缺省的段寄存器。例如：JMP 指令是假设 CS 寄存器，PUSH 和 POP 是假设 SS 是存器，而 MOV 指令是假设 DS 寄存器。

当 assembler 需要去存取（或参考）一个地址，它必须知道哪一个段包含这个地址。使用 ASSUME 伪指令去指定所参考的段或群组地址。文法如下：

ASSUME segregister：seglocation [，segregister：seglocation]

ASSUME dataregister：qualifiedtype [，dataregister：qualifiedtype]

ASSUME register：ERROR [register：ERROR]

ASSUME [register：] NOTHING [，register：NOTHING]

ASSUME register：FLAT [，register：FLAT]

使用 ASSUME 伪指令是告诉 assembler 哪一个段是相关联哪一个段寄存并不是指定处理器。也就是 ASSUME 伪指令只在编译时间（assembly time）发生影响与作用。MASM6.1 会自动把目前的代码段的地址设置给 CS。因此，你不需去包含

ASSUME CS：MY-CODE

在你的程序开头。（如果你要目前的段对应于 CS 的话）。

ASSUME 伪指令可以定义一个段给每一个段寄存器。segregister 可以是 CS，DS，或 SS（FS 和 GS 只在 80386/486 有）。而 seglocation 必定是下列的其中一种：

- 在 SOURCE 文件中用 SEGMENT 伪指令定义的段名称。
- 在 SOURCE 文件中用 GROUP 伪指令定义的群组名称。
- 关键字 NOTHING，ERROR，或 FLAT。
- 一个 SEG 表示式。

关键字（keyword）NOTHING 可以取消目前段的假设。例如 ASSUME NOTHING 取消所有先前用 ASSUME 语句假设的所有寄存器。

通常一个单一的 ASSUME 语句可以在程序的开头就定义了所有的段寄存器。然而你也可以使用 ASSUME 伪指令在程序的任何地方去更改所有段的假设。你可以用下列的语句去避免寄存器的使用：

ASSUME SegRegister：ERROR

当你用简化 directive 去建立数据段，assembler 将产生一个 ASSUME CS：ERROR，有效地避免任何指令或标记出现在数据段中。

11. 定义段群组（Segment Groups）

一个群组是多个段集合物（在 16 位模式不超过 64KB 的段）。一个程序可以由这个群组的开头寻址各段里程序码或数据项目。

一个群组可以让你将许多分离的逻辑段内的不同数据结合在一个段中称为群组（group）。因为使用群组就不必再载入段寄存器的值，可节省你原本要存取各段的数据要再重新载入各段所对应的段寄存器的时间，使你的程序可以使用较少的指令且执行得较快。

一般我们最常使用的群组名称（尤其是 near data）是 DGROUP。在 Microsoft 的段模式，许多段（_DATA，_BSS，CONST 和 STACK）都是结合成一个名为 DGROUP 的群组。Microsoft 高级语言会放置所有 near 数据段在这个群组中。（缺省时连 STACK 也是放在 DGROUP 中）。

.MODEL 伪指令自动会定义一个 DGROUP 群组。而 DS 寄存器正常会指向这个群组(DGROUP) 的开头，而使你能较快速地存取在 DGROUP 里的所有数据。这个群组 directive 语法如下：

name GROUP segment [, segment] …

name 是群组的名称，如 DGROUP（我们最常看到或使用的名称)。而且它可参考此群组先前的定义。例如，你先前已定义了一个群组 MYGROUP，包含 ASEG 和 BSEG 两个段，然而你又加了一个语句如下：

MYGROUP GROUP CSEG

而这样使用是合法的。它会自动将 CSEG 段再加入 MYGROUP 中，而使得 MYGROUP 群组包含了三个段（ASEG，BSEG，CSEG)。

在原始文件中，任何有效的段名称（包含在之前或之后定义的段）都可使用在群组中。但有一个限制是，同一个段名称不能出现在两个群组中。

GROUP 伪指令并不会影响在群组中各个段被载入的顺序。你可以放置任意个 16 位的段在一个群组中，只要所有大小不超过 65536 bytes（64KB)。如果处理器是在 32 位模式，这最大的 SIZE 是 4GB。

附录D　中　　断

中断是一种信号，它是使计算机的中央处理单元暂停正在执行的工作，并转移至一个叫做中断处理程序（Interrupt Handler）或称中断服务程序的特殊程序。这使得硬件能有更好的执行效率和应变能力。

为了有效地处理中断，现代的处理器大多数提供多重的中断种类或层次，CPU 必须在正在执行重要指令时能阻绝中断信号。正在处理一项中断的时候，优先次序相同或较低的中断记号就不能发生作用；同理，如果优先次序较高的中断要求处理时，处理器正在处理的中断信号就会被取代。

中断处理程序必须遵循一个很简单而实在的步骤：

（1）保存系统内部（寄存器、标志以及任何会被处理器更改但是 CPU 不会自动保存的东西）。

（2）中断信号如果被允许在处理程序的操作过程发生，必须加以切断，以免它们产生干扰（这点通常由计算机的硬件自动完成）。

（3）使那些在处理程序操作过程仍被允许发生的中断信号成为有效。

（4）判断中断的原因，针对中断信号采取适当的操作，恢复系统在中断发生前的内容。

（5）重新使在处理程序执行中被切断的任何中断层次成为有效。恢复执行被中断的工作。

1. 如何使用中断

当一个程序或硬件装置需要 CPU 协助时，就会送一个中断的信号或指令给 CPU，表明它需要 CPU 执行特定的工作。通常 CPU 就会停止目前所有的工作，根据中断的性质，去执行一个内存（RAM）中的子程序，叫做‘中断处理程序’(interrup handler) 或称‘中断服务程序’。

在 Intel 80x86 微处理器中，提供了 256 种不同的中断，每一中断由 00h 至 0FFh 的值代表这 256 个中断处理程序的地址存在系统内存最开头的位置，即由 0000：0000H 的中断向量表中每一个中断向量占 4 个 bytes，即将中断值乘 4 就是此中断在中断向量表中的地址。在这个中断向量表中存着此中断处理程序的开头地址，即开始执行的地址。程序设计者将自己设计的中断处理程序的地址，存进中断向量表称为“挂上（hooked）”一个中断，下次任何程序只要用到这个中断号码，新的中断处理程序（Interrupt Service Routine）就会被使用到。只要更改中断向量（中断号码）的内容，就可让它指向另一个新的中断处理程序。

第 0 号服务程序的地址在（00000H～00003H）

第 1 号服务程序的地址在（00004H～00007H）

⋮

第 0FF 号服务程序的地址在（003FCH～003FFH）

前两个 bytes 存储中断服务程序真正所在地址的差距地址 EIP。

后两个 bytes 存储中断服务程序真正所在地址的分段地址 CS。

2. 中断的种类

通常主要可分为三大类：

第一类：CPU 自发的中断

例如 CPU 发生除 0 的运算时，便会执行第 0 号中断，此为 BIOS 提供的中断服务程序。此服务程序位在第 0 号中断向量表中即（00000H～00003H）的地址，会在屏幕上发出“Divide overflow”的警告信息，并将控制权交回 DOS。

第二类：软件中断

也是 ROM BIOS 和 DOS 的服务项目之一，程序通过 INT 指令（int 中断号码）的方式对 CPU 提出要求，软件中断和“CALL”很象，程序设计者不必知道这个中断的服务程序会在什么地方，就连 Compiler、Linker 也不必知道。执行 INT 指令后，CPU 会执行下列工作：

（1）将标志寄存器 Flag 内容推入堆栈内 32 bits。

（2）清除单步标志和中断标志（TF＝0，IF＝0），此时就可禁止其它中断情况发生。

（3）将寄存器 CS 内容推入堆栈，当然高 16 bits 的值没有意义。

（4）将中断号码乘 4 求出中断的地址，将此地址内容的第二个字节存入寄存器 CS 内。

（5）将 EIP（指令寄存器）内容推入堆栈，再将第一个字节存入寄存器 EIP 内，即将 CS：EIP 指向中断处理程序地址。例如利用中断 12H 以中断向量表中的地址 0000：0048H 找到中断处理程序的地址，再将控制权转移给此中断处理程序，如 CPU 就可去执行此中断 12H 的服务程序，此服务程序最后的指令是 IRET。中断回返指令 IRET 会将自堆栈内取回 3 个 32 位的 data（即原来存在堆栈的 CS：EIP 和 Flag 标志值），POP 回各寄存器，再将控制权交回原来的程序。

INT 与 CALL 的差异

使用 call 指令时，会将程序返回地址搬入 stack，而 INT 除了会将程序返回地址推入 stack 中，还会先做一个 pushf 指令，将标志寄存器的内容推入 stack，保留原来的标志寄存器状态。

另外执行 call 后，CPU 在做完将返回地址推入堆栈后，就会直接转到所调用的地址去执行；而执行 INT 指令，则必须先到位于内存（RAM）最前端 00000H（即 0000：0000H）至 003FFH（即 0000：03FFH）的中断向量表；取得中断处理（服务）程序在内存的地址，并根据此地址去执行中断服务程序。

中断程序与一般子程序差异

中断程序与一般子程序差别在于一般子程序是以 RET 指令，直接从堆栈中取回返回地址返回原来的呼程序；而中断程序则是以 IRET 指令返回，与 RET 不同之处在于多做了一个 POPF 指令；由堆栈来复原原来标志寄存器的内容。刚好 INT 指令与中断程序配合，而 CALL 与一般子程序配合。

第三类：硬件中断

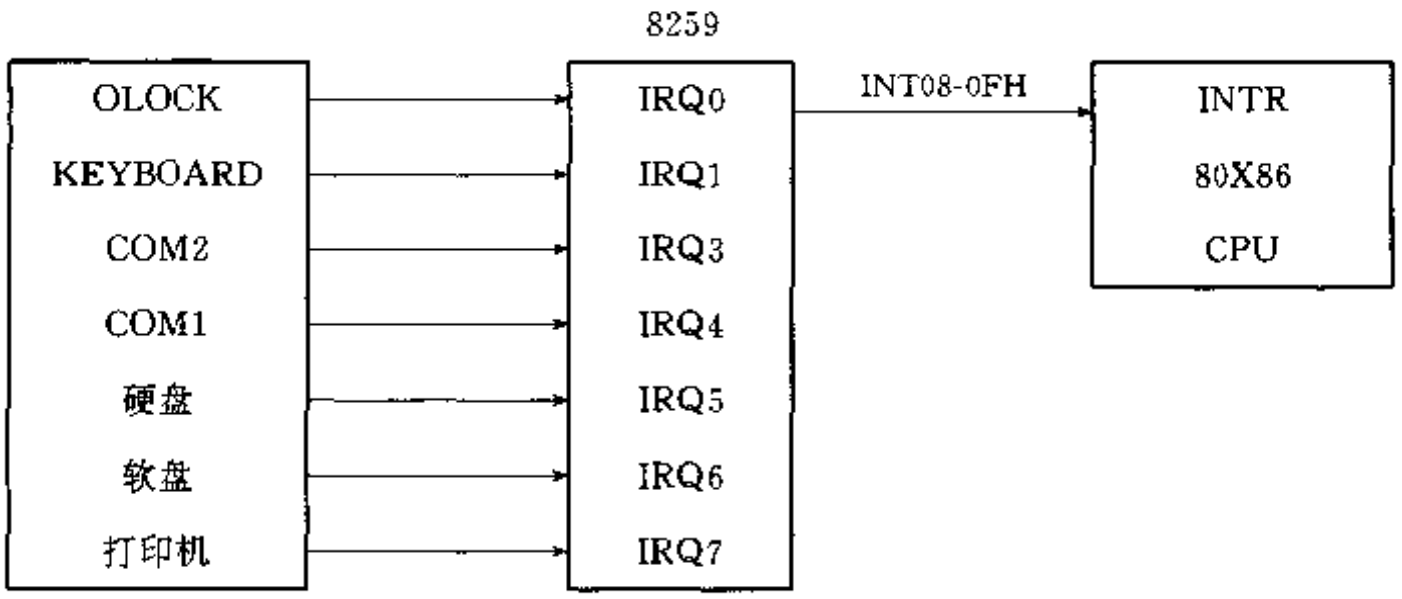
硬件中断是为了回应外围设备所传回的电子信号而发生。PC 上会有一个可程序化的中断控制器（programmable interrupt controller，PIC），即 8259 中断控制器来监管 CPU 外部硬件中断程序。系统上像系统计时器、硬式驱动器、键盘或序列式的通信埠等设备都可以通过硬件中断要求线（IRQ）来产生中断信号，当然这些信号都是由 8259 监管并判断此信号的重要性，再决定是否要中断。每条 IRQ 线都有一个相对的中断值（中断号码）。当某一设备产生中断时，8259 就会将中断值摆到系统数据总线上，以便 CPU 处理。8529 会依不同的中断要

求来分等级。

例如计时器的时序中断最优先，其 IRQ 线为 IRQ 0，其中断号码为 08H。如果在此处理期间内产生一个等级较低的中断，8529 就会延迟至原来中断处理完后，再对 CPU 发出新的中断信号。XT 内有 8 条 IRQ 线。AT、386、486 有 15 条 IRQ 线，而一个 8259 可在同一时间内处理 8 个中断要求，还可和其它的 8529 连接起来，以处理更多的中断，即在 AT 中便可在同一时间内处理 15 个中断，系统在开机时，将 IRQ0 至 IRQ7 设置成中断号码 08H 至 0FH。

系统上有一个特别的中断，称为‘不可遮蔽中断’(Non-Maskable interrupt，NMI)，其中断号码为 02H。(当硬件发出中断要求时，必须 IF＝1 时，CPU 才会执行中断。CLI 指令可使 IF＝0，当然还是由 8259 控制)。此中断无法用 CLI 指令来停止其中断发生。这个 NMI 有绝对优先权，所以一定是很严重或很紧急的情况才会发生。例如硬件的内存有误(故障)，RAM 会产生一个 NMI 中断；CPU 就会将控制权转至中断 02H 的处理程序上。通常是产生一个“PARITY CHECK”的信息出来。

```
/ * 当然程序设计者应该不会用到此中断 * /
```



外部硬件装置设计线路时，便是通过“中断要求线路”(interruprrequest lines，IRQS) 向 8259 达到中断要求。8259 评估要求中断状况，若没问题就将 CPU 上的中断辨识线 (interrupt-acknowledge line，INTA) 设为高电位。CPU 检查 IF 是否为 1 (即中断标志是否为 1)。假如 IF＝1，8259 再把中断号码放入系统数据总线、CPU 数据总线，CPU 收到号码便执行相对应的中断处理程序。在执行 IRET 指令前，中断程序一定要送出一个“中断结束”的序列 (end-of-interrupt，EQI) 给 8259，使得其它中断可以被接受处理。

当一个程序更改了中断向量后，一定得在交还控制权给 DOS 之前，恢复它们原先的向量值，程序用自己的中断程序取代原先的硬件中断一定得作指定、结束的必要工作，重新打开中断，送 EOI 的信号给 8259。如果不遵照这些规则，往往会出现莫名奇妙的执行结果。新的中断程序取代原来的程序；原先的向量一定要存起来，以便以后可以再放回来。直接在中断向量表中修改向量值，将使程序在多工 (Multitasking)、多用户 (Multiuser) 环境、或未来版本的 MS-DOS 中执行。推荐使用 MS-DOS 内置的函数、INT 21H、函数 25H 功能 (AH＝25H 设置中断向量、函数 35H 功能 (AH＝35H，取得中断向量)、函数 31H 功能 (AH＝31H，结束并且维持常驻状态)，以保持与 DOS 的兼容性。

3. 中断优先权

Intel 的 CPU 具有内置的顺序来决定中断的优先顺序。优先权最高的是一些内部的中断

线像是除以零；即优先权的大小由中断号码来决定。也就是说，中断 00H 是具有最大优先权的中断，即 0FFh 永远不能打断其它的中断以进行自己这个中断。而硬件中断（08～0FH）的优先权都比软件中断（INT）来得大。

在 MS-DOS 下外部硬件中断的处理程序，有一些颇为严格的限制：

（1）在中断处理过程中，MS-DOS 绝不可被硬件中断处理程序调用。这是不被允许的。

（2）中断处理程序必须在取得控制权后立即打开中断信号（STI，以免瘫痪了其它装置或破坏了系统时钟的准确性）。

（3）在执行 IRET 指令之前，中断处理程序必须发出中断结束（EOI）指令给 8259，否则具有相同或较低层次的中断信号无法发生作用。

内部硬件中断（00H～07H）或系统例外状况（像 Ctrl-C 等）。若使用新的处理程序来取代 MS-DOS 原来的处理程序，必须小心，以免毁掉中断向量表或造成系统不稳定。处理规则注意事项：

（1）使用 INT 21H 功能 25H 来修改中断向量，以便通知 DOS。

（2）不要直接修改中断向量表，否则将来在多工操作系统中将产生问题。

（3）假如不打算常驻，在程序执行之前保存中断向量目前内容，执行结束恢复原先保存下来的值。

（4）如果打算常驻 RAM，则最好使用 INT 21H，功能 31H 以便保留适当的 RAM 供你的处理程序用。

（5）如果是处理硬件中断，让中断信号被关掉的时间与处理程序所花时间最少。注意，即使在中断信号利用 STI 指令再打开后，如果中断信号是通过 8259，那么相同或较低优先次序的中断信号仍然会被阻切断。

附录 E　MASM 6.11 保留字

此附录列出所有被 MASM 所认识的保留字。主要的种类是：

Operators and symbols

Registers

Operators and directives

Processor instructions

Coprocessor instruction

在 MASM 6.11 中的保留字（可使用在 .8086 模式；default），都可以使用在较高级的 CPU 模式中。若要使用其它种类的保留字，比如“Special Operands for 80386”，需要 .386 模式或较高等级。

你也可以根据对 OPTION 伪指令设置 NOKEYWORD 选项而使附录中你所指定的保留字 DISABLE。一旦 DISABLE 之后，此保留字可以任何方式使用，就如同用户所定义的符号（也就是使这个字成为一有效的识别字）。比如，如果你要从由 MASM 所认识的保留字集合中除去 STR 指令、MASK 操作数和 NAME 伪指令，可加入下列的语句至你的程序中：

OPTION NOKEYWORD：〈STR MASK NAME〉

在附录中的保留字有以（*）表示者，是 MASM 5.1 以后新加入的保留字。

1. Operands and Symbols

在这一章节中所列出来的字对某些伪指令而言是操作数，它们对编译程序有特别的意义。在前半部列表中的字不是保留字，它们可任意被使用如同一般的识别字，不会影响它在伪指令中当操作数的使用。编译程序会从上下文来解释它的使用。

即使前半部列表中的字不是保留的，但是他们也不应该被定义成宏或宏函数。如果执意这样，编译程序并不认识它们也不会有警告信息出现。

ABS	LARGE	NOTHING
ALL	LISTING *	NOTPUBLIC *
ASSUMES	LJMP *	OLDMACROS *
AT	LOADDS *	OLDSTRUCTS *
CASEMAP *	M510 *	OS _ DOS *
COMMON	MEDIUM	PARA
COMPACT	MEMORY	PRIVATE *
CPU *	NEARSTACK *	PROLOGUE *
DOTNAME *	NODOTNAME *	RADIX *
EMULATOR *	NOEMULATOR *	READONLY *
EPILOGUE *	NOKEYWORD *	REQ *
ERROR *	NOLJMP *	SCOPED *
EXPORT *	NOM510 *	SETIF2 *

EXPR16 *	NONE	SMALL
EXPR32 *	NONUNIQUE *	STACK
FARSTACK *	NOOLDMACROS *	TINY
FLAT	NOOLDSTRUCTS *	USE16
FORCEFRAME	NOREADONLY *	USE32
HUGE	NOSCOPED *	USES
LANGUAGE *	NOSIGNEXTEND *	

下面的操作数是保留字且不分大小写。

$	DWORD	PASCAL
?	FAR	QWORD
@B	FAR16 *	REAL4 *
@F	FORTRAN	REAL8 *
ADDR *	FWORD	REAL10 *
BASIC	NEAR	SBYTE *
BYTE	NEAR16 *	SDWORD *
C	OVERFLOW? *	SIGN? *
CARRY? *	PARITY? *	STDCALL *
SWORD *	TBYTE	WORD
SYSCALL *	VARARG *	ZERO? *

2. Special Operands for the 80386/486

FLAT *	NEAR32 *	FAR32 *

3. Predefined Symbols

不像大部分 MASM 保留字，predefined symbols 是分大小写的。

@CatStr *	@Environ *	@Model *
@code	@fardata	@SizeStr *
@CodeSize	@fardata?	@stack *
@Cpu	@FileCur *	@SubStr *
@CurSeg	@FileName	@Time *
@data	@InStr *	@Version
@DataSize	@Interface *	@WordSize
@Date *	@Line *	

4. Registers

AH	BP	CR2	DI	DR3	EAX	EDX
AL	BX	CR3	DL	DR6	EBP	ES

AX	CH	CS	DR0	DR7	EBX	ESI
BH	CL	CX	DR1	DS	ECX	ESP
BL	CR0	DH	DR2	DX	EDI	FS
GS	SP	ST	TR4 *	TR6		
SI	SS	TR3 *	TR5 *	TR7		

5. Operators and Directives

.186	.ERRDIFI	.TYPE.
.286	.ERRE	.UNTIL *
.286C	.ERRIDN	.UNTILCXZ *
.286P	.ERRIDNI	.WHILE *
.287	.ERRNB	.XALL
.386	.ERRNDEF	.XCREF
.386C	.ERRNZ	.XLIST
.386P	.EXIT *	ALIAS *
.387	.FARDATA	ALIGN
.486 *	.FARDATA?	ASSUME
.486P *	.IF *	CATSTR
.8086	.LALL	COMM
.8087	.LFCOND	.COMMENT
.ALPHA	.LIST	DB
.BREAK *	.LISTALL *	DD
.CODE	.LISTIF *	DF
.CONST	.LISTMACRO *	DOSSEG
.CONTINUE *	.LISTMACROALL *	DQ
.CREF	.MODEL	DT
.DATA	.NO87 *	DUP
.DATA?	.NOCREF *	DW
.DOSSEG *	.NOLIST *	ECHO *
.ELSE *	.NOLISTIF	ELSE
.ELSEIF *	.NOLISTMACRO *	ELSEIF1
.ENDW *	.REPEAT *	ELSEF2
.ERR	.SALL	ELSEIFB
.ERR1	.SEQ	ELSEIFDEF
.ERR2	.SFCOND	ELSEIFDIF
.ERRB	.STACK	ELSEIFDIFI
.ERRDEF	.STARTUP *	ELSEIFE
.ERRDIF	.TFCOND	ELSEIFIDN

ELSEIFIDNI	IFIDN	POPCONTEXT *
ELSEIFNB	IFIDNI	PROC
ELSEIFNDEF	IFNB	PROTO *
END	IFNDEF	PTR
ENDIF	INCLUDE	PUBLIC
ENDM	INCLUDELIB	PURGE
ENDP	INSTR	PUSHCONTEXT *
ENDS	INVOKE *	RECORD
EQ	IRP	REPEAT *
EQU	IRPC	REPT
EVEN	LABEL	SEG
EXITM	LE	SEGMENT
EXTERN *	LENGTH	SHORT
EXTERNDEF *	LENGTHOF *	SIZE
EXTERN	LOCAL	SIZEOF *
FOR *	LOW	SIZESTR
FORC *	LOWWORD *	STRUC
GE	LROFFSET *	STRUCT *
GOTO *	LT	SUBSTR
GROUP	MACRO	SUBTITLE *
GT	MASK	SUBTTL
HIGH	MOD	TEXTEQU *
HIGHWORD	.MSFLOAT	THIS
IF	NAME	TITLE
IF1	NE	TYPE
IF2	OFFSET	TYPEDEF *
IFB	OPATTR *	UNION *
IFDEF	OPTION *	WHILE *
IFDIF	ORG	WIDTH
IFDIFI	%OUT	
IFE	PAGE	

6. Processor Instructions

处理器指令是不分大小写的。

7. 8086/8088 Processor Instructions

AAA	CLD	DEC	IRET	JL
AAD	CLI	DIV	JA	JLE
AAM	CMC	ESC	JAE	JMP

AAS	CMP	HLT	JB	JNA
ADC	CMPS	IDIV	JBE	JNAE
ADD	CMPSB	IMUL	JC	JNB
AND	CMPSW	IN	JCXZ	JNBE
CALL	CWD	INC	JE	JNC
CBW	DAA	INT	JG	JNE
CLC	DAS	INTO	JGE	JNG
JNGE	LEA	MOV	RCR	STC
JNL	LES	MOVS	RET	STD
JNLE	LODS	MOVSB	RETF	STI
JNO	LODSB	MOVSW	RETN	STOS
JNP	LODSW	MUL	ROL	STOSB
JNS	LOOP	NEG	ROR	STOSW
JNZ	LOOPE	NOP	SAHF	SUB
JO	LOOPEW *	NOT	SAL	TEST
JP	LOOPNE	OR	SAR	WAIT
JPE	LOOPNEW *	OUT	SBB	XCHG
JPO	LOOPNZ	POP	SCAS	XLAT
JS	LOOPNZW *	POPF	SCASB	XLATB
JZ	LOOPW *	PUSH	SCASW	XOR
LAHF	LOOPZ	PUSHF	SHL	
LDS	LOOPZW *	RCL	SHR	

8. 80186 Processor Instructions

BOUND	INSB	OUTS	POPA
ENTER	INSW	OUTSB	PUSHA
INS	LEAVE	OUTSW	PUSHW *

9. 80286 Processor Instructions

ARPL	LSL	SIDT	SMSW	VERR
LAR	SGDT	SLDT	STR	VERW

10. 80286 and 80386 Privileged-Mode Instructions

GLTS	LIDT	LMSW
LGDT	LLDT	LTR

11. 80386 Processor Instructions

BSF	LFS	PUSHD *	SETNAE	SETP

BSR	LGS	PUSHFD	SETNB	SETPE
BT	LODSD	SCASD	SETNBE	SETPO
BTC	LOOPD *	SETA	SETNC	SETS
BTR	LOOPED *	SETAE	SETNE	SETZ
BTS	LOOPNED *	SETB	SETNG	SHLD
CDQ	LOOPNZD *	SETBE	SETNGE	SHRD
CMPSD	LOOPZD *	SETC	SETNL	STOSD
CWDE	LSS	SETE	SETNLE	
INSD	MOVSD	SETG	SETNO	
IRETD	MOVSX	SETGE	SETNP	
IRETDF *	MOVZX	SETL	SETNS	
IRETF *	OUTSD	SETLE	SETNZ	
JECXZ	POPAD	SETNA	SETO	

12. 80486 Processor Instructions

BSWAP *	INVD *	WBINVD *
CMPXCHG *	INVLPG *	XADD *

13. Instruction Prefixes

LOCK	REPE	REPNZ
REP	REPNE	REPZ